USTL FLANDRES ARTOIS — LIFL — U.A 369 du C.N.R.S

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° d'ordre : 1177

THÈSE

présentée à

L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

pour obtenir le titre de

DOCTEUR DE 3ème CYCLE

INFORMATIQUE

par

Francesco DE COMITÉ

SIMULATION LINEAIRE DE SYSTEMES DE REECRITURE

Thèse soutenue le 22 Juin 1984 devant la Commission d'Examen

Membres du Jury :

Président :	C. CARREZ
Rapporteur :	M. DAUCHET
Examinateurs :	A. ARNOLD
	B. COURCELLE
	G. COMYN

NIVERSITE DES SCIENCES
T TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie végétale
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie-Physique
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DHAINAUT André	Biologie animale
M. DOUKHAN Jean Claude	Physique du solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie numérique
M. GRANELLE Jean jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie organique
M. LACOSTE Louis	Biologie végétale
M. LAVEINE Jean Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHOMME Jean	Chimie organique biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie physique
M. LUCQUIN Michel	Chimie physique
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphéri
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie générale
M. PETIT Francis	Chimie organique
M. POUZET Pierre	Modélisation - Calcul scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Mle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean Paul	Géométrie et topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie animale
M. CATTEAU Jean pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean jacques	Biologie
Mle DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBRABANT Pierre	Géologie appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du solide - Cristallographie
M. DERIEUX Jean Claude	Microbiologie
Mle DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DEVRAINNE Pierre	Chimie minérale
Mme DHAINAUT Nicole	Biologie animale
M. DHAMELINCOURT Paul	Chimie physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUBUS Jean Paul	Spectrométrie des solides

M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériq
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie physique
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEVRE Christian	Pétrologie
Mle LEGRAND Denise	Algèbre
Mle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes electroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée

M. PERTUZON Emile	Physiologie animale
M. PONSOLLE Louis	Chimie physique
M. PORCHET Maurice	Biologie animale
M. POSTAIRE Jack	Informatique industrielle
M. POVY Lucien	Automatique
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean CLaude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROY Jean Claude	Psychophysiologie
Mme SCHWARZBACH Yvette	Géométrie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
M. STAROSWIECKI Marcel	Informatique
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. THERY Pierre	Systèmes électroniques
M. THIEBAULT François	Sciences de la terre
M. THUMERELLE Pierre	Démographie - Géographie Humaine
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du Travail
M. TURREL Georges	Spectrochimie infrarouge et Raman
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. WACRENIER Jean Marie	Electronique
M. WALLART Francis	Spectrochimie infrarouge et Raman
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	Géologie générale
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. WERNER Georges	Informatique théorique
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Simulation linéaire de systèmes de réécriture

****

**

*

Table des matières

INTRODUCTION

Les arbres (ou termes) permettent de représenter simplement et de façon parlante un grand nombre de structures telles que : expressions algébriques, syntaxe de phrases ou de programmes, hiérarchie entre objets, représentation des connaissances dans les systèmes experts etc...

Afin de les étudier et de pouvoir manipuler cette structure, les chercheurs ont été amenés à créer un certain nombre d'outils spécifiques (automates, transducteurs d'arbres, attributs).

En particulier, les systèmes de réécriture (sdr), permettent d'exprimer de manière naturelle des règles de calcul sur les arbres.

L'étude des sdr a conduit à un certain nombre de résultats, dont les plus puissants concernent les sdr linéaires, c'est-à-dire ceux où l'on ne teste ni n'engendre aucune égalité explicite de sous-arbres.

Dans le but de circonscrire la portée des sdr linéaires relativement aux sdr généraux, nous introduisons deux notions de simulation (forte et faible), d'un sdr par un autre.

Nous montrons que tout sdr peut être simulé, même fortement par un sdr linéaire, mais que l'on perd certaines propriétés très importantes (confluence ou terminaison finie).

Afin de calculer le prix payé pour la linéarisation, nous introduisons la notion de complexité d'un système de réécriture, notion qui correspond intuitivement au nombre de "passages" nécessaires pour dériver complètement un arbre.

Nous sommes alors à même d'évaluer le fossé séparant les sdr linéaires des autres.

INTRODUCTION AUX SYSTEMES DE REECRITURE

- DEFINITIONS

- QUELQUES PROPRIETES CONNUES

Un alphabet gradué est un ensemble de couples de la forme (lettre, arité) où la lettre s'appelle aussi symbole fonctionnel, et l'arité est un entier. Dans la pratique, nous ne considérons que les alphabets gradués tels qu'à chaque lettre ne corresponde qu'une arité.

par exemple : Σ { a , b , $\bar{a}$ } est un alphabet gradué et l'arité de a est égale à 2, celle de b à 3. (Chaque lettre de arité n peut être vue comme un opérateur à n arguments).

L'arité de $\bar{a}$ est égale à o : on dit alors que $\bar{a}$ est une constante de Σ. Les variables de substitution sont les éléments d'un ensemble dénombrable X de lettres de arité o, tel que $X \cap \Sigma = \emptyset$.

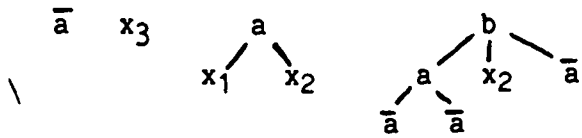
Une variable marquera dans un arbre un endroit où l'on peut placer un arbre quelconque.

Un arbre est une structure bâtie à partir d'un alphabet gradué Σ et d'un ensemble de variables X, vérifiant les propriétés suivantes :

1) Dans l'arbre, il existe une seule lettre n'ayant pas de prédécesseurs (la racine de l'arbre)

2) Si l'arité de la racine est égale à n, elle a n successeurs qui sont eux aussi des arbres.

exemple : si Σ = { b , a , $\bar{a}$ } et $X = \{x_i / i \in N\}$
alors les structures suivantes sont des arbres :



Remarques :

1) On représente habituellement les arbres avec leurs racines en haut.

2) On appelle feuilles les symboles sans successeurs (donc les constantes et les variables).

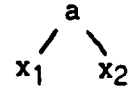
3) Lorsque dans un arbre plusieurs variables portent le même nom, elles ne peuvent être remplacées (on dira aussi instanciées) que par le même arbre. On peut exprimer ainsi des conditions d'égalité ou des duplications d'arbres. (cela sera utilisé dans les systèmes de réécriture).

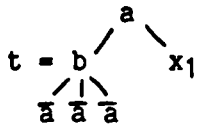
Par analogie avec les arbres généalogiques, on parlera du père d'un noeud (le prédécesseur de celui-ci), des fils d'un noeud et plus généralement des ascendants et descendants d'un noeud.

On dira d'un arbre t qu'il est <u>filtré</u> par un arbre g si t peut être obtenu à partir de g en instanciant, et en renommant éventuellement, certaines variables dans g.

exemples :

1° $\Sigma = \{ a^{(2)}, b^{(3)}, \bar{a}^{(0)} \}$ $\quad X = \{x_i / i \in N\}$

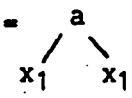
$g =$ 

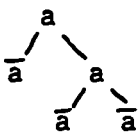
, $t =$ 

t est filtré par g si l'on remplace (on parle plutôt de substitution) :

x_1 par  et x_2 renommé x_1.

2° Par contre :

si $g =$ 

$t =$ 

t n'est pas filtré par g, car les deux sous-arbres de t correspondant aux deux occurences de x_1 dans g ne sont pas égaux.

Dans g, les deux fils de  doivent être égaux : on dira que g <u>n'est pas linéaire</u>.

On peut dire, de manière informelle, que t est filtré par g si t est un cas particulier de g.

A la suite de ces quelques notions, il est désormais naturel et facile d'introduire les <u>systèmes de réécriture</u>.

Ceux-ci peuvent être considérés comme des schémas de calcul indiquant très précisement quelles sont les transformations d'arbres autorisées ainsi que les conditions à vérifier avant d'opérer ces transformations. Plus précisément, un système de réécriture est défini par :

1) L'ensemble d'arbres qu'il permet de manipuler (autrement dit, l'alphabet gradué sur lequel il opère)

2) Un ensemble fini de paires orientées d'arbres, qu'on notera $g \to d$, telles que toute variable apparaissant dans d apparaisse dans g :

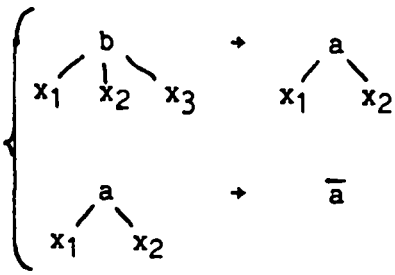
les règles de réécriture

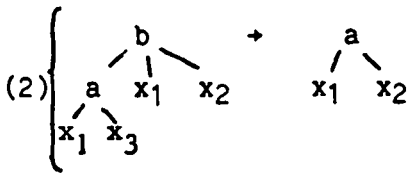
(On comprend aisément cette condition : toute règle est appelée à être appliquée dans un arbre quelconque : g fixe les conditions dans lesquelles la règle sera applicable, les variables apparaissant dans g servent donc à préciser le contexte nécessaire à l'application de la règle).

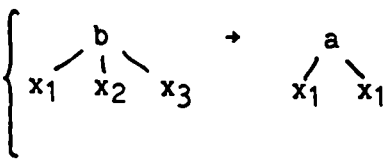
Exemple :

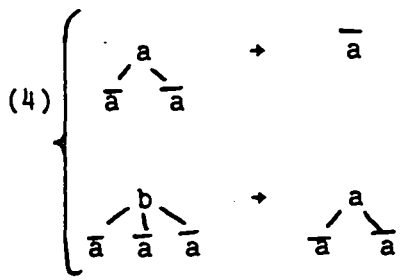
Soit $\Sigma = \{ b , a , \overline{a} \}$ $\quad X = \{x_i / i \in N\}$

Les quatre ensembles de règles suivants sont des systèmes de réécriture sur $T(\Sigma)$ (l'ensemble des arbres bâti à partir de $\Sigma \cup X$) :

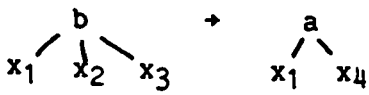
(1)


(2)


(3)


(4)


par contre :



n'est pas une règle de réécriture, car on s'autoriserait alors à engendrer sous a un arbre quelconque (par instanciation de la variable x_4, qui n'appartient pas au contexte de départ).

Remarques :

La présence de deux variables x_1 dans la règle composant le système de réécriture (2) précise que cette règle ne sera applicable que si les deux arbres instanciant les x_1 sont égaux.

On voit là la puissance des systèmes de réécriture : ils permettent d'exprimer de façon claire et finie des conditions d'égalité nécessaires à l'application de la règle.

Si l'on considère maintenant le système de réécriture (3), on voit que là, c'est dans le membre droit de la règle qu'apparaissent deux variables x_1. Il ne s'agit plus ici d'un test d'égalité préalable à l'application de la règle, mais de l'expression d'une duplication d'un sous-arbre, sous certaines conditions exprimées par le membre gauche de la règle. Les règles composant les systèmes de réécriture (2) et (3) seront dites respectivement non linéaire à gauche et non linéaire à droite.

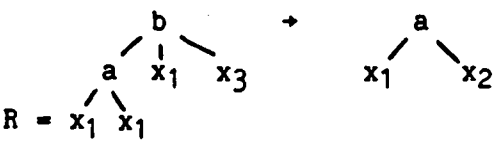
Les systèmes de réécriture (1) et (4), où dans chaque membre (droit ou gauche) de chaque règle, les variables apparaissent au plus une fois seront dits linéaires.

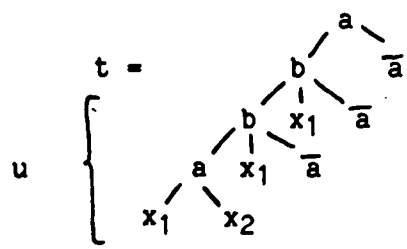
L'application d'une règle de réécriture g → d dans un arbre t correspondra à la suite d'actions suivante :

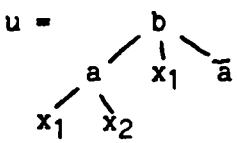
a - trouver un sous-arbre u de t qui soit filtré par g.

b - remplacer u par d, en instanciant les variables de d par les sous-arbres calculés lors du filtrage de u par g.

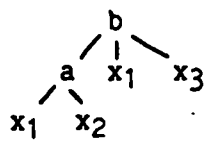
exemple : $\Sigma = \{ b \quad a \quad \bar{a} \}$ $\qquad X = \{x_i / i \in N\}$





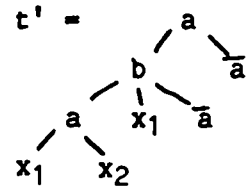


est filtré par $g =$

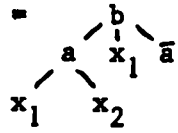


(en instanciant x_3 par $\bar{a}$).

On dérive donc t en t' =



dans lequel on peut isoler le sous-arbre u' =



, filtré par g

En effet, tout arbre de T (Σ) est, soit égal à a, soit de la forme $\begin{matrix} a^n \\ | \\ y \end{matrix}$

Dans le premier cas, aucune règle de R n'est applicable.

Dans le second cas, $t = \begin{matrix} a^n \\ | \\ y \end{matrix}$ peut aussi s'écrire $\begin{matrix} a \\ | \\ a^{n-1} \\ | \\ y \end{matrix}$.

En instanciant x par $\begin{matrix} a_{n-1} \\ | \\ y \end{matrix}$, on filtre t par $\begin{matrix} a \\ | \\ y \end{matrix}$, on peut alors appliquer la règle de R.

Donc, en n dérivations, $\begin{matrix} a^n \\ | \\ y \end{matrix}$ se dérive en y

Ainsi, pour tout arbre, le nombre de dérivations qui y sont applicables est borné par sa taille.

R est donc noethérien.

par contre ; $\Sigma = \{ \begin{matrix} a \\ | \end{matrix}, \bar{a} \}$ $\qquad R = \begin{matrix} a \\ | \\ x \end{matrix} \rightarrow \begin{matrix} a \\ | \\ a \\ | \\ x \end{matrix}$

n'est pas noetherien (clairement, chaque dérivation ajoute une nouvelle possibilité de dérivation)

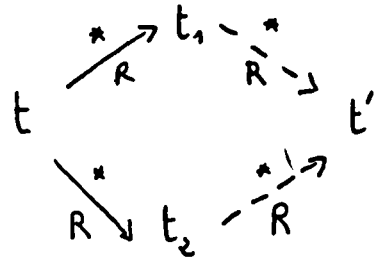
Remarquons que lorsqu'un système de réécriture est noetherien, tout arbre possède une forme normale (dans l'exemple, si $t = \begin{matrix} a^n \\ | \\ y \end{matrix}$, alors sa forme normale est y) mais celle-ci n'est pas forcément unique, comme le montre l'exemple suivant :

$\Sigma = \{a_1, a_2, a_3\}$ $\qquad R = \begin{cases} a_1 \rightarrow a_2 \\ a_1 \rightarrow a_3 \end{cases}$

soit $t = a_1$ t peut se dériver en a_2 ou a_3 qui sont ses deux formes normales.

2) On dit d'un système de réécriture R qu'il est confluent si, lorsqu'un arbre t peut se dériver en deux arbres distincts t_1 et t_2, alors il existe un arbre t' tel que t_1 et t_2 se dérivent par R en t'.

On peut figurer cette propriété sur le schéma suivant, où les flèches pleines représentent les hypothèses et les flèches pointillées les conclusions :



Exemples :

1) Le système précédent n'était pas confluent, car aucune règle de R ne permet plus de dériver a_2 et a_3.

2) Si on lui rajoute l'une des deux règles $a_2 \to a_3$ ou $a_3 \to a_2$, il devient alors confluent.

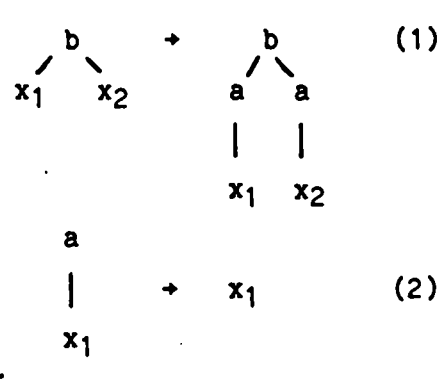
On dira qu'un système de réécriture est <u>localement confluent</u> si le passage de t à t_1, et celui de t à t_2, se font par application d'une seule règle de ce système.

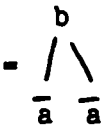
Lorsqu'un système de réécriture est confluent, tout arbre a au plus une forme normale.

3) Un système de réécriture est dit acyclique si partant d'un arbre quelconque t, aucune chaîne de dérivations ne ramène à t.

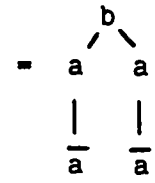
$\Sigma = \{ b \overset{/\backslash}{,} \quad a \overset{|}{,} \quad \bar{a} \}$ $\qquad X = \{x_i / i \in N\}$

$R =$



n'est pas acyclique. Il suffit de considérer l'arbre t =  on peut le

dériver par application de la règle (1) en :

$t' =$  , puis appliquer la règle (2) aux deux fils de  , retrouver l'arbre t.

Sans être équivalentes, ces trois propriétés (terminaison finie, confluence, acyclisme) sont très liées ainsi :

1° Un sdr noethérien est acyclique

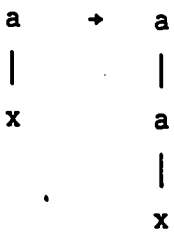
2° Un sdr noethérien et localement confluent est confluent

3° Dans un sdr noethérien et confluent, tout terme possède une forme normale unique.

Mais :

1° Un sdr acyclique n'est pas forcément noethérien.

Exemple : $\Sigma = \{a , \bar{a}\}$ $X = \{x_i / i \in N\}$

$R =$ 

Ce système de réécriture est acyclique (la taille d'un arbre croît strictement à chaque dérivation), mais n'est pas noethérien.

Malheureusement, ces propriétés ne sont pas décidables dans le cas général. On est donc amené à donner des conditions suffisantes pour les vérifier.

En ce qui concerne la terminaison finie, Knuth et Bendix ont remarqué que si l'on pouvait trouver un ordre bien fondé (<) sur les arbres tel que pour toute règle $g \to d$ d'un sdr R, on ait $d < g$, alors R est noethérien. Eux-mêmes en définissent alors un, consistant principalement à doter chaque symbole fonctionnel d'un poids, et à donner alors des règles permettant de comparer deux arbres en regardant leur poids respectif.

Dans le même esprit, un certain nombre d'auteurs ont introduit d'autres ordres sur les arbres, les principaux étant l'ordre récursif sur les chemins de Dershowitz, les ordres de simplification et de décomposition (P. Lescanne, F. Reinig). Ce dernier ordre a d'ailleurs la propriété très interessante, du point de vue de l'implémentation, d'être incrémental :

c'est-à-dire qu'il permet de définir l'ordre adéquat sur les lettres de l'alphabet gradué Σ pour que les règles soient bien orientées, sans modifier les relations d'ordre déjà calculées.
D'autre part, des extensions des systèmes de réécriture permettant de prendre en compte des équivalences d'arbres (correspondant par exemple, à des règles exprimant la commutativité) ont conduit à définir les notions de terminaison finie et de confluence modulo un ensemble d'équations. On peut encore alors utiliser certains des ordres précédents afin d'orienter ces équations conformément à ces ordres, les transformant alors en règles de réécriture.

La confluence, qui dans le cas général n'est pas non plus décidable, a pu être ramenée, dans certains cas, à l'observation de la locale confluence sur certains couples d'arbres : les paires critiques.

Celles-ci, introduites par Knuth et Bendix, sont obtenues par superposition des membres gauches de règles. On obtient un couple d'arbres (p,q), correspondant aux deux dérivations possibles. Si, pour chacune de ces paires, on peut trouver un arbre t, tel que p et q se dérivent en t, et si le système est noétherien, on montre alors qu'il est confluent.

Les résultats décrits précédemment sont considérablement renforcés quand les systèmes de réécriture considérés sont linéaires.

Par exemple, G. Huet obtient une condition suffisante de confluence pour les théories équationnelles (ensembles de règles de réécriture et d'équations entre arbres), pourvu que le système de réécriture associé soit linéaire à gauche, cette dernière contrainte étant nécessaire, comme il l'illustre sur un contre-exemple.

Par contre, d'autres résultats ont été prouvés sur les sdr linéaires par suite de la méthode employée lors de leur démonstration, sans que cette condition de linéarité ne semble nécessaire. C'est le cas de Guttag, Kapur et Musser, qui obtiennent une condition suffisante pour qu'un sdr soit acyclique. Suivant une démarche proche de celle de Knuth et Bendix pour la confluence, ils introduisent la notion de paire dérivée, qui est un couple d'arbres obtenu par superposition du membre gauche d'une règle avec le membre droit d'une autre.

Définissant ensuite OC(R) (Overlap Closure), l'ensemble de ces paires dérivées, les auteurs montrent que si R est globalement fini (i.e : tout terme n'a qu'un nombre fini de descendants), linéaire à gauche ou linéaire à droite, alors R est noetherien ssi OC(R) ne contient pas de règle de la forme $g \to g$.

Ils remarquent aussi que la condition de linéarité imposée à R paraît peu naturelle et conjecturent qu'elle peut être levée.

Présentation
de notre travail

***

**

Notre travail a pour objectif de bien situer les sdr linéaires par rapport aux autres, et surtout préciser dans quelle mesure le travail accompli par un sdr non linéaire peut être effectué par un sdr linéaire (même si celui-ci est très compliqué) : c'est de là qu'est partie l'idée de simulation.

En définissant celle-ci, nous voulions voir quelles sont les propriétés du système de départ qui sont conservées et celles qui sont perdues. Les résultats montrent que des propriétés comme la terminaison finie ou la confluence sont nécessairement perdues en cas de simulation forte. Cette démarche nous a permis, dans un premier temps (lors de la construction des systèmes simulant) de par la difficulté rencontrée pour linéariser, de mesurer intuitivement la puissance des sdr non linéaires.
Nous l'avons ensuite mesurée grâce à la définition de la complexité.
Nous introduisons deux notions de simulation.
Informellement, on dira qu'un sdr R' simule un sdr R si R' peut faire tout ce que fait R'. Eventuellement, et ce sera le cas pour les deux notions de simulation que nous utiliserons, R' peut être plus "puissant" que R', (dans le sens où certaines dérivations dans R' ne correspondront à aucune dérivation dans R).

La simulation forte est la plus naturelle :

Définition : Soient R, R' deux sdr définis respectivement sur $T(\Sigma)$, $T(\Sigma')$, tels que $\Sigma' \supset \Sigma$

On dira que R' simule fortement R si :

$\forall (u,v) \in T(\Sigma)^2$: i. $u \xrightarrow[R]{*} v \Rightarrow u \xrightarrow[R']{*} v$

ii. Si $u \in T(\Sigma)$ et $u \xrightarrow[R']{+} v$, alors il existe $v_1 \in T(\Sigma)$ tel que :

$$\left.\begin{array}{l} u \xrightarrow[R']{+} v_1 \xrightarrow[R']{*} v \\ \text{ou} \quad u \xrightarrow[R']{*} v \xrightarrow[R']{*} v_1 \end{array}\right\} \text{et } u \xrightarrow[R]{+} v_1$$

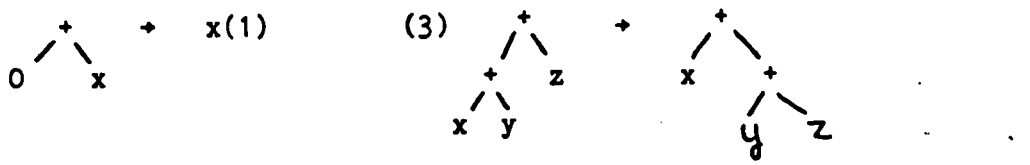
Le point i signifie qu'à toute dérivation dans R correspond une dérivatio dans S.

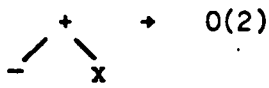
Le point ii signifie que toute dérivation dans S débutant par un arbre d $T(\Sigma)$ peut, soit se prolonger, soit se raccourcir en une dérivation corres pondant à une dérivation dans R. (On exprime ainsi le fait que Σ' est u alphabet intermédiaire, et $T(\Sigma')$ un ensemble d'arbres auxiliaires pour l simulation).

Exemple :

Soit $\Sigma = \{ +, \bar{\ }, 0, a, b, c \}$

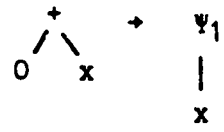
et R défini sur Σ par les règles de réécritures suivantes :

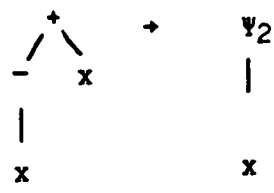


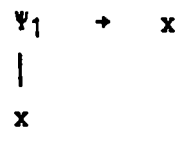


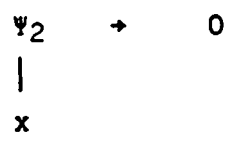
(R exprimant les axiomes des groupes)

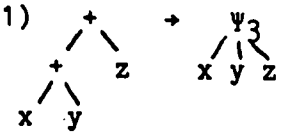
Il est simulé fortement par R', défini sur $\Sigma' = \Sigma \cup \{\Psi_1, \Psi_2, \Psi_3\}$ par :

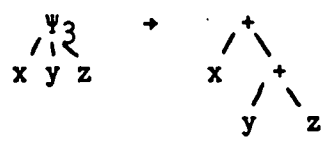
(1.1) 

(2.1) 

(1.2) 

(2.2) 

(3.1) 

(3.2) 

Dans cet exemple, on peut considérer que les nouveaux arbres "mémorisent" la règle en cours d'application, ainsi que les variables utilisées.

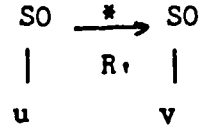
Remarque : Il est capital que R' soir défini sur un alphabet élargi. Les nouveaux symboles introduits sont en fait des "variables de contrôle" permettant de savoir à tout moment où nous en sommes dans la simulation.

La simulation faible, bien que plus contraignante, est plus simple à manipuler, et sans doute à mettre en oeuvre.

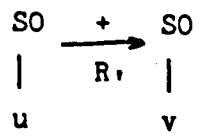
On considère alors des arbres marqués au sommet, cette marque pouvant être ôtée lorsque certaines ou toutes les dérivations possibles dans l'arbre ont été effectuées.

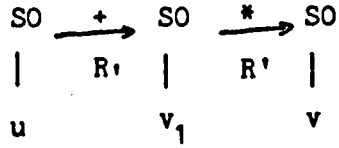
Plus formellement :

Définition : On dira que R' simule faiblement R (sous les hypothèses de la définition précédente) si :

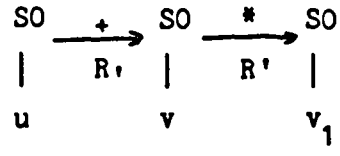
i. $\forall (u,v) \in T(\Sigma)^2 \quad u \xrightarrow[R]{*} v \Rightarrow$ 

(où SO est le marquage de sommet)

ii. $\forall \; u \in T(\Sigma)$  $\Rightarrow \exists\, v_1 \in T(\Sigma)$ tel que



ou bien



et $u \xrightarrow[R]{+} v_1$

Exception faite du marquage des sommets, cette définition est tout à fait similaire à celle de la simulation forte.

Nous montrerons cependant que les deux ont des propriétés très différentes. Les exemples et la mise en oeuvre de la simulation faible sont assez longs et laborieux. Ils font l'objet des chapitres II, III, IV.

Le premier résultat, et sans doute le plus important, concernant ces simulations est énoncé par le théorème suivant :

Théorème :

(1) Il existe un algorithme qui, à tout sdr R associe un sdr linéaire R simulant R sur les termes clos. (Il s'agit en fait de deux algorithmes l'un pour la simulation faible, l'autre pour la simulation forte).

(2) Dans le cas où R' simule fortement R, il faut nécessairement choisir de conserver soit le caractère noethérien, soit la confluence de R. (Il n'est pas possible de garder simultanément ces deux propriétés). Un contre exemple illustrera clairement ce second point.

La simulation d'un sdr par sdr linéaire nous a obligé à résoudre deux problèmes principaux.

- Le premier ('local') de simuler une règle non linéaire par un sdr linéaire : c'est la partie principale de l'algorithme de linéarisation.
- Le second ('global') de contrôler étroitement les simulations de chaque occurence de règles, afin qu'il n'y ait pas d'interférences entre les règles durant une simulation. (Le sous-arbre où s'effectue une simulation doit être isolé du contexte).

Ce dernier point est assez technique. Informellement, nous pouvons considérer qu'il est résolu de la manière suivante :

a) Des "pointeurs" (en fait, de nouveaux symboles fonctionnels) seront présents dans l'arbre à dériver. Ils seront nécessaires pour qu'une règle soit applicable. On leur permet de parcourir l'arbre de manière déterministe. On peut ainsi gérer leur parcours de l'arbre, et savoir à coup sûr quelle sera la prochaine règle à appliquer.

b) Des indices (fixes ou portés par les pointeurs) mémoriseront le numéro de la règle en cours de simulation.

c) Le contexte sera "gelé", isolant ainsi le sous-arbre en cours de simulation de dérivation.

On peut considérer, pour plus de clarté, que lorsqu'une simulation est amorcée, toutes les règles appliquées dans l'arbre ont pour unique but d'achever cette simulation.
Nous nous contenterons, dans cette introduction, de donner une idée intuitive de ce qu'est la simulation d'une règle par une règle linéaire à droite, sans détailler les structures de contrôle (nécessaires mais lourdes à manier).
Nous supposerons seulement ici qu'elles existent et sont efficaces.

Considérons la règle (non linéaire à droite) suivante :

(En raison de l'impasse faite ici sur les structures de contrôle, nous considererons implicitement que l'on pointe sur a).

Il s'agira ici d'engendrer linéairement l'égalité représentée par

On accomplira ce travail grâce à un algorithme utilisant les primitives suivantes :

a) Parcourir un arbre t jusqu'à un noeud v.

Ce parcours est déterministe : on choisira dans cet exemple celui correspondant à la règle "de la main gauche" : Explorer le sous-arbre, puis la racine, et enfin le sous-arbre droit.

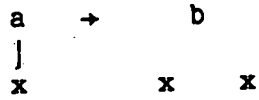
b) Parcours inverse

c) "Déposer un caillou" : marquer un noeud v.

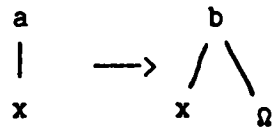
d) "Reprendre un caillou" : retirer la marque portée par un noeud v.

L'algorithme aura pour tâche de dupliquer, à la place du fils droit de b, la valeur de son fils gauche. On peut le décrire par algorithme suivant :

Simulation linéaire de



Début

Appliquer la règle linéaire 

Aller chercher la valeur de la feuille la plus à gauche dans x, et l
recopier en Ω ;

Tant que la copie n'est pas terminée faire

Parcourir l'arbre jusqu'au premier noeud non marqué ;

Marquer le noeud et mémoriser sa valeur ;

Parcourir l'arbre jusqu'à l'endroit de la copie ;

Recopier ;

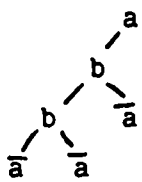
fait

Retirer les marques ;

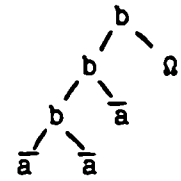
fin

Nous illustrerons cette description (nécessairement très schématique) su
l'exemple suivant.

Soit l'arbre :

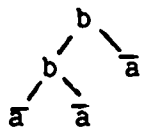


il se transforme tout d'abord en :

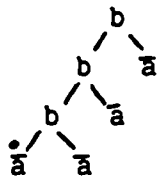


où a ∈ Σ, Ω ∈ Σ' Σ.

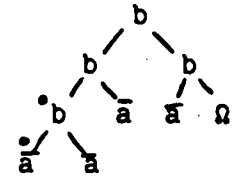
La première étape consiste à marquer, puis aller recopier la feuille l
plus à gauche de l'arbre :



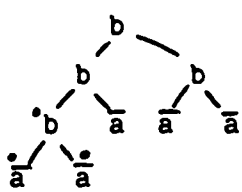
On obtient alors :



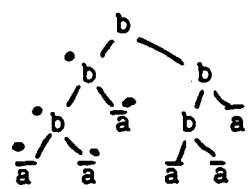
la seconde étape recopie :
le b le plus interne
(engendrant ainsi à nouveau
Ω).



La troisième étape nous conduira alors à l'arbre suivant :



et nous obtenons,
à la sortie du tant que :



Il ne reste plus alors qu'à ôter les marques pour terminer la simulation.
Cet exemple appelle plusieurs remarques :

(i) : Une marque spécifique au sommet de l'arbre permet au pointeur de copie (qui remonte l'arbre) de "savoir" quand il doit commencer à chercher le noeud du sous-arbre droit où il doit déposer son information.

(ii) : Le mode de parcours de l'arbre est tel qu'il faut copier alternativement une feuille ou un noeud interne : une simple connaissance du niveau, dans le sous-arbre modèle, du noeud à recopier (calculable lors de la recherche du premier noeud non marqué) permet de savoir à quel endroit déposer cette information.

(iii) : Tous les tests et primitives utilisés s'expriment à l'aide de sdr et de configuration de l'arbre au voisinage du pointeur.

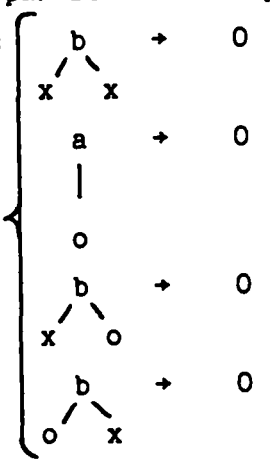
Conservation de la confluence et de la terminaison finie :

La simulation utilisée dans l'exemple, de par sa définition, conserve la confluence éventuelle du système de réécriture simulé.

En utilisant une technique voisine, on peut choisir de conserver la propriété de terminaison finie au détriment de la confluence (très grossièrement, il suffit de ne pas effacer les marques).

Nous ne nous apesantirons pas ici sur ce cas, plutôt anecdotique. Il est intéressant, par contre, de voir qu'on ne peut pas simuler fortement un sdr R confluent et noethérien par un sdr R' linéaire confluent et noethérien. Ce résultat est illustré facilement par le contrexemple suivant :

Soit $\Sigma = \{ b, a, \overline{a}, 0 \}$ et R défini par :



On voit que pour $t \in T(\Sigma)$:

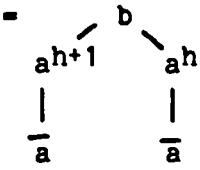
$t \xrightarrow[R]{*} 0$ si un b a deux fils jumeaux ou si $\#(0,t) \neq 0$

$t \xrightarrow[R]{o} t$ sinon

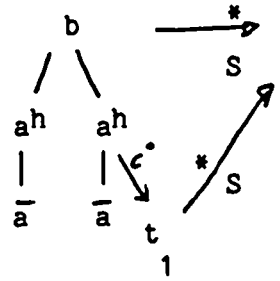
R est noethérien. Soit S simulant R et h > sup { hauteur des membres gauches de règles de S.

Puisque S a un nombre de règles fini, ce sup existe et est fini.

Alors il existe une règle de S applicable à t_o =



En effet :



i étant une règle de S $g_i \to d_i$

Deux cas se présentent :

1) $g = a^j$ (x) $\to d$ avec $j < h$. La règle est donc applicable à t_o

(même conclusion si x est remplacé par a).

2) $g = a^i(x)\ a^j(y)$ $i, j < h$

la règle est ici encore applicable à t_o

Il existe donc une règle de S applicable à t_o. On a donc le schéma suivant :

$t_o \xrightarrow{g_i \to d_i} t' \xrightarrow[o\ S]{*} t_o$, puisque S simule R et $t_o \xrightarrow{o} t_o$ dans R.

S appliqué à t_o possède donc un cycle, il n'est donc pas noethérien.

Remarquons cependant que ce cycle est induit par la présence d'une non linéarité gauche dans le système de départ. (La simulation de la non linéarité à droite conserve l'aspect noethérien).

C'est pour éviter cette perte de propriétés que nous introduisons la notion de simulation faible.

Grossièrement, l'idée est la suivante : on prend un arbre dont un sommet est marqué. A partir de ce sommet, on crée un pointeur que l'on promènera dans l'arbre, afin qu'il permette l'application de règles dans son voisinage. Chaque fois que le sous-arbre repéré par le pointeur "ressemblera" à un membre gauche de règle, on "essaiera" d'appliquer cette règle (principalement dans le cas d'une règle non linéaire à gauche, où il faudra vérifier l'égalité de certains sous-arbres).

Simultanément, on mémorise le numéro de la règle qu'on tente d'appliquer ; on peut donc, par marquage des arbres, mémoriser les règles inapplicables et éviter de nouvelles tentatives qui créeraient des cycles.
Lorsqu'aucune règle n'est plus applicable dans l'arbre (on peut le "savoir" grâce à une configuration d'arbre), on efface alors le pointeur et le marquage du sommet.

Sans marquage au sommet, il est impossible de savoir si l'arbre est en forme normale. On peut alors indéfiniment essayer d'appliquer des règles, ce qui crée donc des cycles.

Nous utilisons ensuite les résultats concernant les simulations linéaires pour les appliquer au cas des théories équationnelles.

Une théorie équationnelle est la donnée d'un ensemble d'équations entre arbres $t_i = t'_i$

Pourvu que pour tout i, $V(t_i) = V(t'_i)$, on peut associer à toute théorie équationnelle E un système de réécriture R_E dont les règles sont formées par dédoublement des équations de E. C'est-à-dire que si $t_i = t'_i \in E$, alors les deux règles $t_i \to t'_i$, $t'_i \to t_i$ appartiennent à R. (Et R ne contient pas d'autres règles).

A partir de ce système R_E (qu'on dira <u>symétrique</u>, car si une règle $u \to v \in R_E$, alors $v \to u \in R_E$), nous allons construire un sdr linéaire le simulant.

Nous utiliserons ici la simulation forte, car l'aspect noethérien n'a plus d'importance, et la définition doit être symétrique.

Il ne sera pas nécessaire de simuler entièrement R_E : nous ne prendrons que la moitié des règles de ce sdr (de telle façon qu'aucun couple $(t_i \to t'_i, t'_i \to t_i)$ ne s'y retrouve), et simulerons linéairement ce demi-système par un sdr dont les règles seront <u>réflexives</u>.

Une règle $u \to v$ sera dite reflexive si, dans n'importe quel arbre elle peut être sans ambiguïté appliquée dans le sens $v \to u$.
En clair, si l'on peut appliquer $v \to u$, c'est qu'auparavant on a appliqué au même endroit $u \to v$.

Une sophistication des méthodes de simulation définies précédemment perme
d'obtenir un tel système :

Il suffit, pour chaque règle de $\underline{RE}_2$ simulée, de créer les règles

a) Marquant les sommets des sous-arbres à comparer ou recopier

b) Envoyant les pointeurs d'un sommet à l'autre.

Pratiquement, on s'arrangera pour n'avoir à simuler que des règles linéai
res à gauche (autant que faire se peut).

On obtiendra ainsi la linéarisation à gauche des règles inverses en renver
sant l'algorithme de linéarisation à droite.

Par exemple :

Soit l'équation $a(x) = b(x,x)$

La règle $a(x) \to b(x,x)$ a été simulée précédemment

Pour simuler $b(x,x) \to a(x)$, il suffira de "faire marcher l'a
gorithme à l'envers".

Pour un arbre de la forme $b(t,t)$, on aboutira alors finalement à $a(t)$.

Par contre, pour $b(t,t')$, $t \neq t'$, il deviendra impossible, à partir d'un
certain moment, de continuer : la seule possibilité serait de "revenir
arrière".

Ce système simulant R' composé de règles réversibles permet ainsi de déf
nir le sdr $R'^{-1} = \{u \to v / v \to u \in R'\}$.

Le système $R' \cup R'^{-1}$ simule donc R_E linéairement.

On obtient donc le résultat suivant :

Théorème :

Il existe un algorithme qui, à partir d'une théorie équationnelle E, construit une théorie équationnelle E_ℓ linéaire simulant fortement E (pourvu que toute équation de E soit complète : $V(t_1) = V(t')$)
1

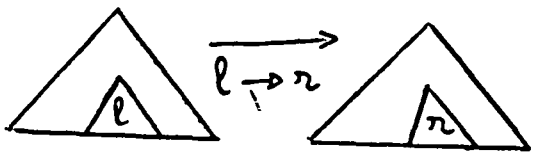
En fait, E_ℓ est la théorie équationnelle associée au sdr symétrique $R' \cup R'^{-1}$.

Les exemples utilisés dans cette introduction, bien que très simplifiés, montrent clairement qu'un sdr non linéaire, est en général simulé par un sdr linéaire très compliqué.

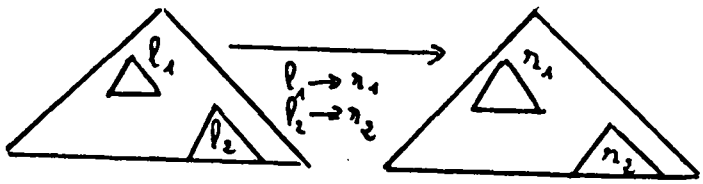
Nous avons voulu donner une consistance à cette notion, en définissant la complexité d'un système de réécriture.

La profondeur d'une transformation $t \xrightarrow[R]{*} u$ correspond au nombre de passage minimal nécessaire à la dérivation de t en u.
(Au cours d'un passage, on applique en une seule fois toutes les règles applicables indépendemment).
Nous montrons que la profondeur d'une transformation est aussi la longueur maximale des chaines de règles imbriquées.

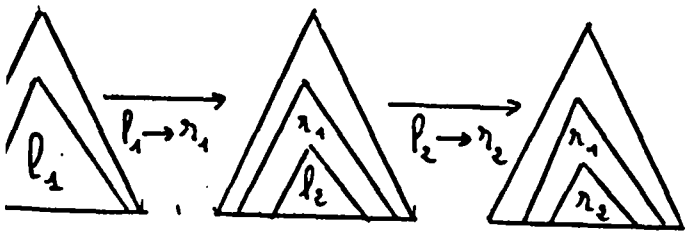
Exemples :



transformation de profondeur 1



profondeur 1



profondeur 2
(il faut attendre que la première règle soit appliquée avant de pouvoir appliquer la seconde).

Nous pouvons alors définir $\mathcal{C}(R)$, la complexité d'un sdr, comme étan la borne supérieure des profondeurs de transformation.

Cela nous permet d'établir les deux résultats suivants :

Théorème 1 :

Par un sdr non linéaire de complexité 5, on peut engendrer toute for récursivement énumérable.

Ce résultat est obtenu à partir de la technique utilisée par Dauch et J. Mongy : il suffit pratiquement de réécrire leur preuve sous forme système de réécriture.

Théorème 2 :

Un sdr linéaire de complexité finie quelconque conserve la reconnai sabilité.

Lorsqu'on sait que la classe des forêts reconnaissables est une tou petite classe où tout est décidable, on mesure immédiatement le goufre puissance séparant les sdr non linéaires des sdr linéaires.

Plan :

Nous introduisons, après quelques rappels et définitions, les systèm de réécriture contrôlés, qui permettront dans les phases ultérieures d simulations, d'isoler les points d'applications des règles, de localiser d'enchaîner les différentes primitives de simulation.

Nous construisons ensuite respectivement, pour une règle de réécritu quelconque, les sdr la simulant linéaire à droite, puis à gauche. (Le pr mier correspondant à une copie, le second à un test d'égalité).

Nous relions ensuite ces deux simulations, afin de simuler linéair ment, à droite comme à gauche, un sdr quelconque.

Nous appliquons ensuite une technique similaire pour simuler linéa rement des théories équationnelles.

Le dernier chapitre introduit la notion de complexité, permettant mesurer la distance existant entre les sdr linéaires et les autres.

Les chapitres II à V, définissant les simulations doivent être regardés comme la description d'un programme linéarisant un sdr quelconque.

Les simulations qui y sont décrites sont plus complexes que celles illustrées dans les exemples de cette introduction : c'est qu'elles décrivent, à un niveau proche d'un langage machine, la suite exacte des opérations à effectuer, suite que nous n'avons fait qu'esquisser dans nos exemples.

CHAPITRE I

Définitions

DEFINITIONS

A. Rappel de notions classiques :

Remarque :

Sans altérer la généralité de ce qui suit, et dans un souci de simplification, nous considèrerons que deux lettres du même alphabet gradué ne peuvent avoir le même label.

On peut alors poser :

Déf. 1 : Un alphabet gradué est la donnée d'un couple (Σ, ar) où Σ est un ensemble dont les éléments s'appellent des lettres (ou symboles fonctionnels) et ar une application de Σ dans $\mathbb{N}$ (arité).

Dans tout ce qui suit, Σ sera de cardinal fini.

Déf. 2 : Soit (Σ, d) un alphabet gradué, on notera

$$\Sigma_i = \{f \in \Sigma / ar(f) = i\}$$

Déf. 3 : On appelle ensemble des variables l'ensemble infini

$$X = \{x_1, x_2 \ldots\} \qquad X_n = \{x_1, \ldots x_n\}, \quad X_o = \emptyset$$

I - Arbres :

Déf. 4 : On appelle ensemble des arbres sur Σ indexés par X_n l'ensemble $T(\Sigma)^1_n$ défini par :

- $\Sigma_o \cup X_n \subset T(\Sigma)^1_n$
- Si $a \in \Sigma_p$ et $t_1, \ldots, t_p \in T(\Sigma)^1_n$ alors $a(t_1, \ldots, t_p) \in T(\Sigma)^1_n$

En particulier $T(\Sigma)^1_o$ représente l'ensemble des arbres sans variables construits sur Σ. (encore appelés termes clos).

Déf. 5 : Un noeud d'un arbre t est une occurence dans t d'une lettre de Σ ou d'une variable

Le label d'un noeud est cette lettre ou cette variable.

Déf. 6 : La taille d'un arbre t, notée $|t|$ est le nombre de ses noeuds.

Déf. 7 : La hauteur (ou profondeur) d'un arbre t est le nombre maximal de neouds séparant la racine d'une feuille. On la notera h(t).

Plus précisemment : si $t = a \in \Sigma_0 \cup X$, $h(t) = 0$

si $t = f(u_1,\ldots u_n)$ $h(t) = 1 + \underset{i\in[1,n]}{\text{Max}}\ (h(u_i))$

Déf. 8 : Un arbre sera dit linéaire si toute variable de X y apparait au plus une fois.

Déf. 9 : On appelle nombre d'occurences d'une lettre f ou d'une variable x dans un arbre t le nombre de fois où cette lettre ou variable apparait dans t.

On le notera $\#(f,t)$ (ou $\#(x,t)$)

Déf. 10 : On notera V(t) l'ensemble des variables $x \in X$ telles que $\#(x,t) \neq 0$.

II – Systèmes de réécriture :

Déf. 1 : Un système de réécriture R défini sur $T(\Sigma)$ est un ensemble de paires dirigées $g \to d$ tel que :

$(g,d) \in T(\Sigma)^2$

$V(d) \subset V(g)$

Déf. 2 : Un terme t appartenant à $T(\Sigma)$ se réduit par R à l'occurence u en un terme t' par la règle $g \to d$ (ce qu'on écrit $t \xrightarrow[(u,g\to d)]{R} t'$)

ssi il existe une substitution s unifiant g et t/u et $t' = t[u \leftarrow s(d)]$.

Déf. 3 : Une règle $g \to d$ est dite :

a. linéaire à gauche si g est linéaire

b. linéaire à droite si d est linéaire

c. linéaire si g et d sont linéaires.

Remarque :

Lorsqu'il n'y a pas d'ambiguïté, on peut omettre dans l'écriture

$t \xrightarrow[(u,g\to d)]{R} t'$

R et/ou $(u, g \to d)$.

Déf. 4 : La relation de dérivation $\xrightarrow{*}$ est la clôture transitive et réflexive de la relation $\longrightarrow$.

Déf. 5 : Un terme t est en forme normale pour R (on dit encore R-irréductible) ssi

$(t \xrightarrow{*} t') \Rightarrow (t = t')$

(i.e : il n'y a plus de règles de R applicables à t).

t' est une R-forme normale de t ssi - $t \xrightarrow[R]{*} t'$

- t' est en forme normale pour R.

Quelques propriétés des systèmes de réécriture :

Déf. 1 : Un système de réécriture R est noethérien (ou à terminaison finie) s'il n'existe pas de chaîne infinie de réductions.

C'est-à-dire qu'il n'y a pas de suite de la forme :

$$t_0 \to t_1 \to \ldots \to t_n \to \ldots$$

Déf. 2 : R est confluent si, pour tout triplet (t, t_1, t_2) $T(\Sigma)$:

$$\left.\begin{array}{l} t \xrightarrow{*} t_1 \\ t \xrightarrow{*} t_2 \end{array}\right\} \Rightarrow \exists\, t' \text{ tel que } \left\{\begin{array}{l} t_1 \xrightarrow{*} t' \\ t_2 \xrightarrow{*} t' \end{array}\right.$$

Déf. 3 : R est dit convergent ssi il est noéthérien et confluent.

Remarque : Si R est convergent, tout terme $t \in T(\Sigma)$ admet une forme normale unique notée $t\,!$.

Déf. 4 : R est localement confluent ssi pour tout triplet (t, t_1, t_2) de $T(\Sigma)$

$$\left.\begin{array}{l} t \longrightarrow t_1 \\ t \longrightarrow t_2 \end{array}\right\} \Rightarrow \exists\, t' \; t_q \left\{\begin{array}{l} t_1 \xrightarrow{*} t' \\ t_2 \xrightarrow{*} t' \end{array}\right.$$

B. Les notions de simulation

Soit R défini sur $T(\Sigma)$, R' défini sur $T(\Sigma')$

Déf. On dira que R' simule R ssi :

$$\forall\, (u,v) \in T(\Sigma) \quad u \xrightarrow[R]{*} v \Rightarrow u \xrightarrow[R']{*} v$$

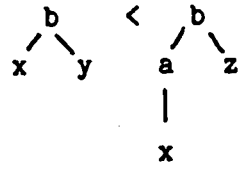
Nous expliciterons et détaillerons cette notion dans les chapitres suivants.

Introduisons maintenant quelques définitions nécessaires à la simulation.

I – Arbres incompatibles

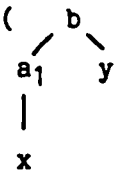
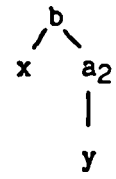
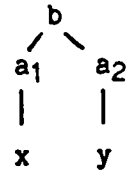
On définit le préordre de filtrage par :

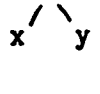
$t \leqslant t'$ ssi t est sous-arbre initial de t'

exemple :  (t est plus général que t')

Déf : On dira que t et t' sont incompatibles si le couple (t,t') n'est pas majoré pour le préordre de filtrage.

exemples :

1. ( , ) est majoré par ex : par 

2. ( , ) n'est pas majoré : les deux arbres sont incompatibles

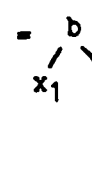
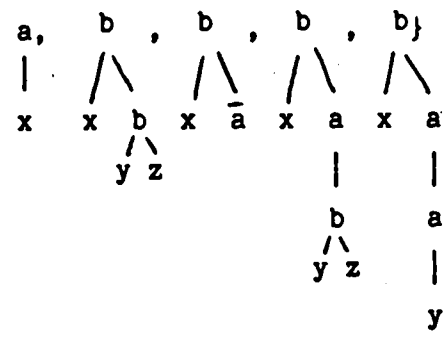
Lemme : $\forall\, t_1$ linéaire, $\exists$ un ensemble fini Incomp (t_1) tel que :

(t_1, t) incompatibles $\Leftrightarrow \exists\, u \in$ Incomp (t_1) tel que $t > u$.

Informellement, Incomp (t_1) représente l'ensemble des "patrons" des arbres incompatibles avec t_1.

Exemple :

$\Sigma = \{ b , a , \bar{a} \}$ 

$t_1 =$  Incomp$(t_1) = \{$ $\bar{a}$,  $\}$

Remarque :

Si t_1 n'est pas linéaire, le lemme précédent est faux.

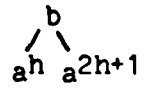
(Incomp(t_1) n'est plus fini).

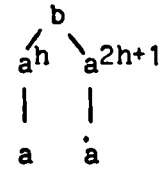
En effet :

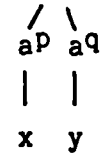
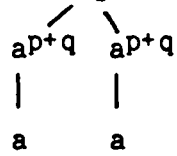
$t_1 =$ 

Supposons que car d(Incomp(t_1)) < + ∞

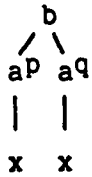
Soit h le sup des hauteurs des arbres de Incomp(t_1).

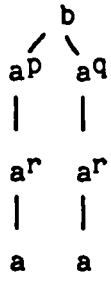
(t_1, ) sont incompatibles donc $\exists$ u $\in$ Incomp(t_1) tel que

$t_2 =$  > u.

- u n'est pas réduit à x, car (x, t_1) sont compatibles
- u n'est pas de la forme  car  > u mais est

compatible avec t_1.

Donc u est de la forme  , avec p, q $\leq$ h

Donc $t_2 =$  avec $\begin{cases} p = r = h \\ q = r = 2h + 1 \end{cases}$

En faisant la différence de ces deux équations, nous obtenons :

$$q - p = h + 1$$

Or p, q $\leq$ h.

Il y a donc contradiction : Incomp(t_1) est donc infini lorsque t_1 n'est pas linéaire.

II – Sens de parcours d'un arbre:

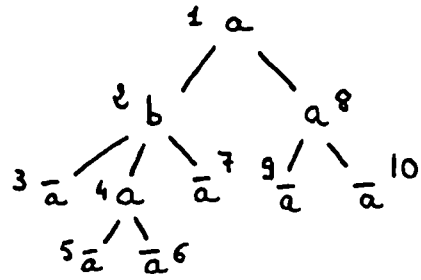
Nous aurons besoin par la suite de définir un sens de parcours d'un arbre. Nous choisissons celui qui consiste à explorer dans l'ordre :

- La racine
- Le sous-arbre gauche
- Le sous-arbre à droite du précédent

On définit ainsi un ordre total sur les noeuds d'un arbre (parcours "de haut en bas et de gauche à droite")que nous appelerons "ordre sens de parcours".

Exemple :

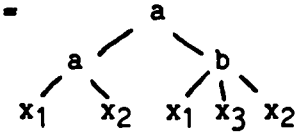
Les noeuds sont numérotés relativement à l'ordre sens de parcours (on a ainsi l'ordre dans lequel ils seront traités).



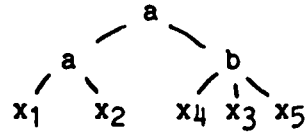
III – Linéarisé d'un terme :

Déf. 1 : On appelle linéarisé d'un arbre t l'arbre t dans lequel on a le cas échéant renommé certaines variables afin de le rendre linéaire.

exemple : t =



linéarisé de t

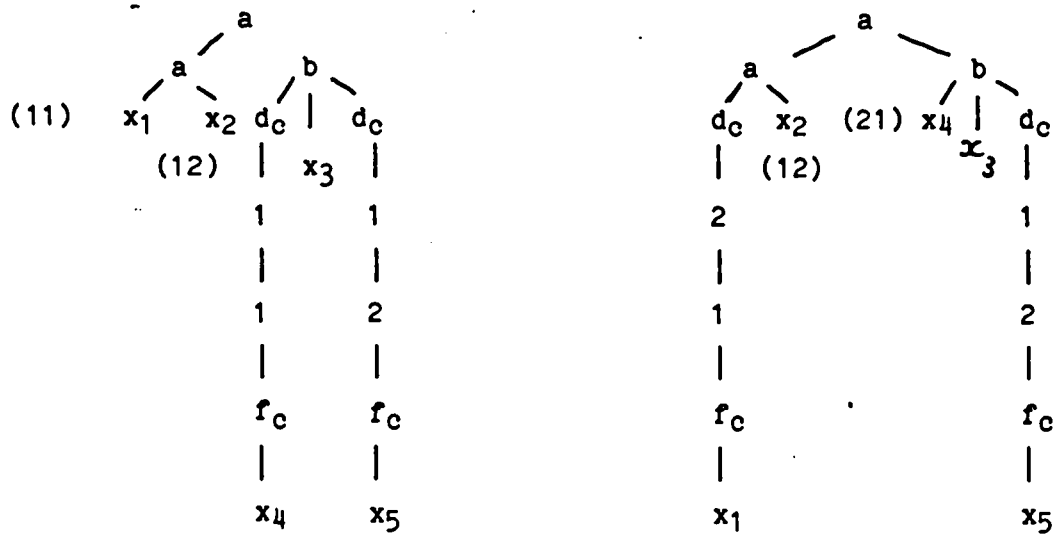


Remarque : Au nom des variables près, le linéarisé de t est unique.

Déf. 2 : On appelle linéarisé gauche indicé de t un linéarisé de t où l'on a intercalé, devant les variables renommées, une suite d'information qui permettra de retrouver le noeud auxquel elles sont égales :

Exemple :

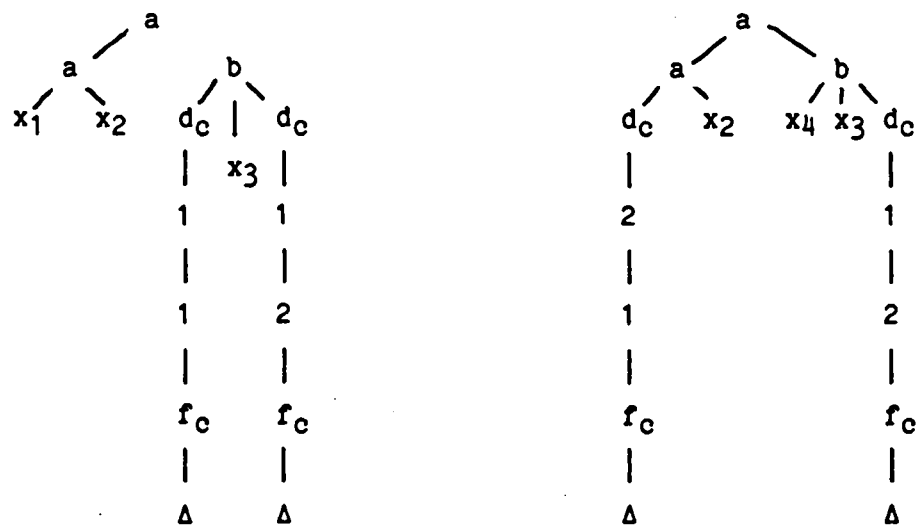
Soit t l'arbre défini précédemment. Les deux arbres suivants sont des linéarisés gauches indicés de t.



(On suppose pour cela que dc, fc, 1,2, ne sont pas éléments de Σ).
En clair, on intercale entre les symboles dc et fc (pour "début du code", "fin du code") le code du noeud de t où se trouve la variable égale à celle suivant le symbole fc.
On remarque que le linéarisé gauche indicé n'est pas unique.

Déf. 3 : On appelle linéarisé droit de t un linéarisé gauche indicé de t dans lequel les variables suivant un symbole fc ont été remplacées par un nouveau symbole fonctionnel Δ.
Exemple :

Pour les deux linéarisés gauches indicés décrits précédemment, nous obtenons les linéarisés droits :



Nous verrons par la suite l'utilité de cette construction (les Δ seront les endroits où effectuer une copie, le modèle devant être cherché au noeud correspondant au code).

Remarques : Un linéaire droit est toujours indicé

On suppose que $\Delta \notin \Sigma$

IV – Pointeurs :

On appelleradans la suite pointeurs un ensemble de symboles fonctionnels n'appartenant pas à Σ, unaires, dont la caractéristique principale sera leur possibilité de se déplacer dans un arbre.

Nous les représenterons sous forme de flèches indicées $\rightarrow_{fac}$, $\rightarrow_{rec}$ etc. (pour plus de clarté nous noterons $\rightarrow$ les pointeurs descendants et $\leftarrow$ les pointeurs remontant).

CHAPITRE II

Structures de contrôle

STRUCTURES DE CONTROLE

Comme nous le verrons lors de sa description, la linéarisation à gauche d'une règle de réécriture consistera en un test d'égalité entre deux sous-arbres.

Afin d'éviter que la simulation d'une telle règle ne crée des cycles, nous avons besoin de contrôler l'application des règles (pour éviter principalement qu'on effectue plusieurs fois le même test d'égalité en cas d'échec de celui-ci).

La nécessité d'un tel contrôle vient aussi du fait que les simulations ne doivent pas interférer entre elles. C'est-à-dire qu'il faudra soit localiser les dérivations soit isoler les sous-arbres où s'effectue une simulation.

Nous définissons deux structures de contrôle correspondant approximativement à chacun de ces choix.

La première est basée principalement sur le marquage du sommet de l'arbre à dériver. A partir de ce marqueur, on enverra un premier pointeur $\rightarrow_{fac}$ défini de telle façon qu'il puisse appliquer dans l'arbre toute règle possible. Revenant ensuite à son point de départ, s'il a engendré une dérivation, on le renvoie dans l'arbre. Sinon, il est remplacé par un second pointeur $\rightarrow_{ob}$ qui lui forcera la première dérivation possible.

Nous empêcherons ainsi la création de cycles et obtiendrons quelques propriétés annexes, dont la plus importante est sans doute une condition de maximalité pour tout arbre, s'exprimant d'une façon exploitable par un système de réécriture.

La deuxième structure de contrôle, utilisera la précédente dans sa phase finale. Le marquage au sommet de celle-ci est alors remplacée par une isolation du contexte de sous-arbres, qui alors seulement seront marqués au sommet.

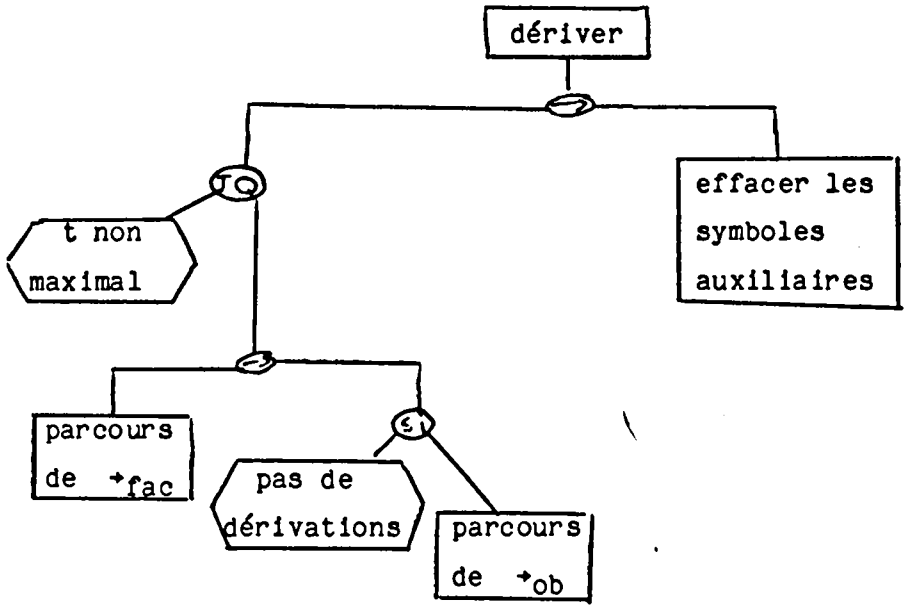
I - Marquage au sommet

Soit R système de réécriture sur $T(\Sigma)$

$t \in T(\Sigma)$

Soit $u = \begin{matrix} SO \\ | \\ t \end{matrix}$ où SO, marqueur de sommet n'appartient pas à Σ

La réécriture de t peut s'exprimer sous la forme suivante :



Remarque : On représente ici le cas extrême où l'on s'oblige à appliquer l'arbre t toutes les règles possibles. Nous montrerons qu'il est possibl d'arrêter cette dérivation à tout instant, sans perdre les propriétés in duites par cette structure de contrôle.

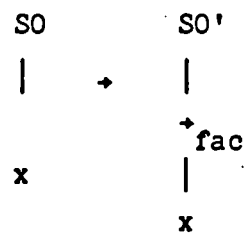
Chaque prédicat et action de cet arbre programmatique s'exprime sous form de configuration d'arbre et de systèmes de réécriture. Nous nous employon à les définir.

Description des primitives

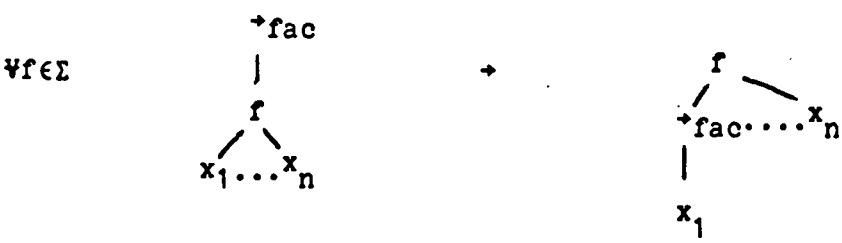
1. parcours de →fac

a. Création du pointeur

Elle est assurée par la règle :

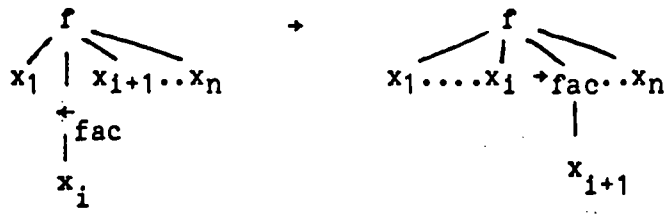


Nous devons créer les règles de récriture lui permettant de parcour l'arbre :



$\forall$ a constante $\in \Sigma$

$\forall f \in \Sigma$ $\forall i < ar(f)$



$\forall f \in \Sigma$

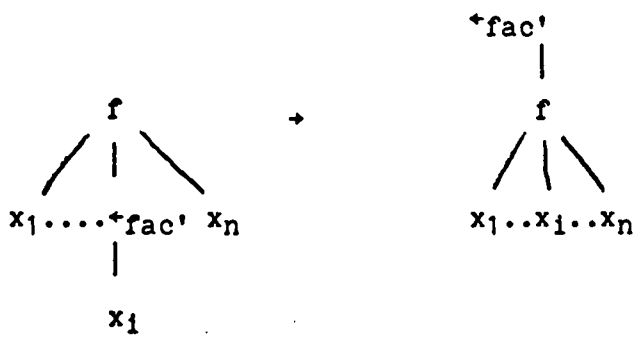
Ces quatres règles permettent à $\rightarrow_{fac}$ de parcourir exhaustivement l'arbre, conformément au sens de parcours défini précédemment.

Il faut maintenant autoriser le pointeur $\rightarrow_{fac}$ à générer une dérivation, ce qui est exprimé par le système de réécriture suivant :

$\forall (g \rightarrow d) \in R$:

et la famille de règles permettant à $\leftarrow_{fac'}$ de remonter au sommet de l'arbre :

$\forall f \in \Sigma$, $\forall_1 \leq i \leq ar(f)$



Le système de réécriture ainsi défini vérifie les deux propriétés suivantes

P1 : Soit $t \in T(\Sigma)^1_0$.

La configuration

$$\begin{array}{c} SO' \\ | \\ \rightarrow_{fac} \\ | \\ t \end{array}$$

signifie exactement que le pointeur $\rightarrow_{fac}$ a parcouru exhaustivement t sans engendrer de dérivation

Démonstration :

- Puisque le pointeur est revenu à son point de départ et étant donné la façon dont il parcourt l'arbre, il est clair qu'il a parcouru entièrement t.

- Lorsqu'il engendre une dérivation, $\rightarrow_{fac}$ se transforme en $\rightarrow_{fac'}$, qui lui ne peut, pour le moment, se transformer.

Donc, dans la configuration finale, $\rightarrow_{fac}$ n'a pas engendré de dérivation.

P2 : La configuration

$$\begin{array}{c} SO' \\ | \\ \rightarrow_{fac'} \\ | \\ t \end{array}$$

signifie que $\rightarrow_{fac}$ a permis une dérivation (et une seule) lors de son parcours de l'arbre

Démonstration :

- Puisque $\rightarrow_{fac}$ s'est transformé en $\rightarrow_{fac'}$, une dérivation au moins été effectuée, qui a engendré $\rightarrow_{fac'}$.

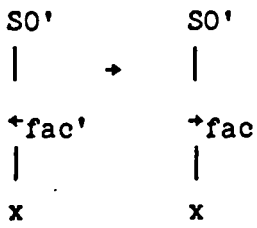
- Ce dernier pointeur, remontant directement jusqu'à SO' ne peut être à l'origine d'une dérivation.

Celle-ci est donc unique.

Nous obtenons de cette façon un test (c'est la condition de si de l'algorithme nous permettant de savoir si l'on peut relancer $\rightarrow_{fac}$ ou l'on doit se servir du pointeur $\rightarrow_{ob}$.

Cette alternative est exprimée par les deux règles de réécriture :

$$\begin{array}{c} SO' \\ | \\ \rightarrow_{fac} \\ | \\ x \end{array} \rightarrow \begin{array}{c} SO' \\ | \\ \rightarrow_{ob} \\ | \\ x \end{array}$$



2. Parcours de $\rightarrow_{ob}$:

Définitions préliminaires :

D1 : Soit R système de réécriture sur un alphabet gradué Σ.

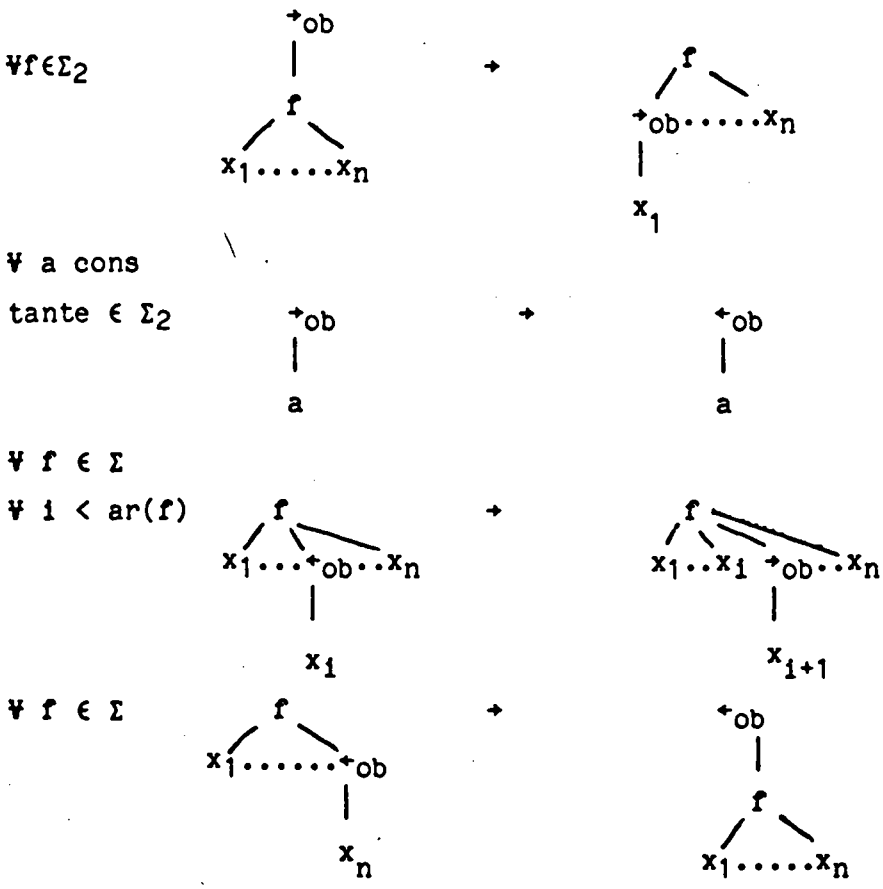
On peut partitionner Σ en deux ensembles Σ_1, Σ_2 tels que toute lettre de Σ_1 soit sommet de membre gauche d'une règle de R.

$f \in \Sigma_1 \iff \exists\, (g \rightarrow d) \in R \,/\, f = g/_\varepsilon$

D2 : Pour tout $f \in \Sigma_1$, on peut définir l'entier p_f représentant le nombre de règles de R dont f est le sommet du membre gauche.

$\forall f \in \Sigma_1 \quad p_f = \text{card}\{(g \rightarrow d) \in R / f = g/_\varepsilon\} > 0$

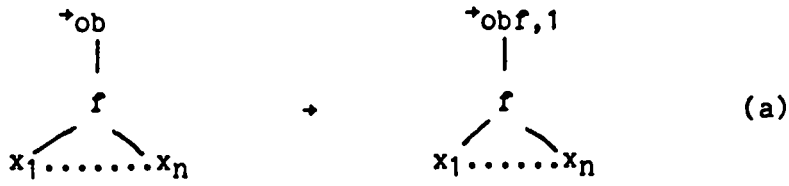
On crée le système de réécriture suivant :



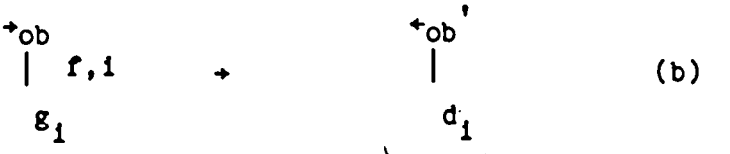
Ce système permet à $\rightarrow_{ob}$ de parcourir l'arbre tant qu'aucune application de règle n'est possible. Voyons maintenant les règles permettant à $\rightarrow_{ob}$ de forcer la première dérivation licite qu'il rencontre.

Soit $\{g_1,\ldots g_{pf}\}$ l'ensemble des membres gauches de règles dont le somme est f.

$\forall f \in \Sigma_1$ (a)

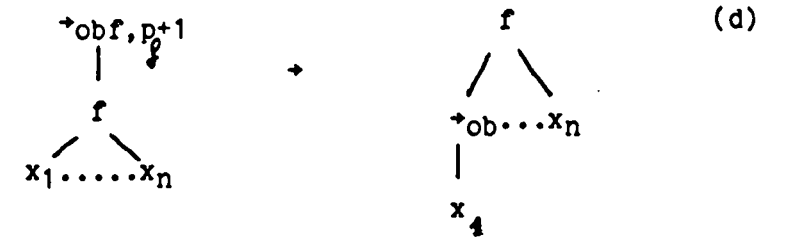


$\forall f \in \Sigma_1$
$\forall i \leq pf$ (b)

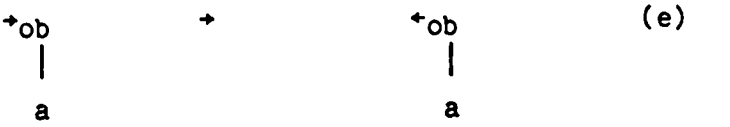


$\forall f \in \Sigma_1$
$\forall i \leq pf$
$\forall t \in Incom(g_i)$ (c)

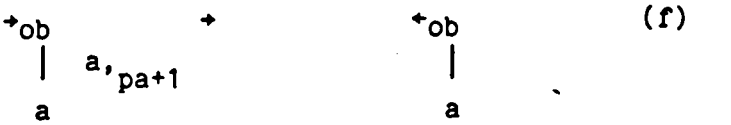
$\forall f \in \Sigma_1$ (d)



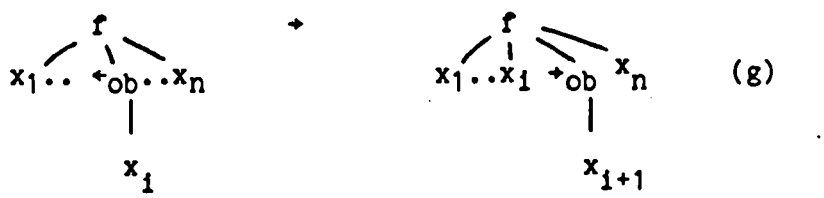
$\forall a \in \Sigma_2$ (e)



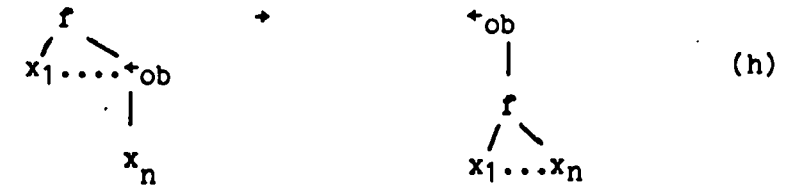
$\forall a \in \Sigma_1$ (f)



$\forall f \in \Sigma$
$\forall i < ar(f)$ (g)



$\forall f \in \Sigma$ (h)



Avant de nous interesser au rôle de ces règles de réécriture, remarquons que si R et Σ sont finis, il en est de même pour ce système de réécriture. On voit clairement que le pointeur $\rightarrow_{ob}$ enclenchera la première dérivation possible qu'il rencontrera, et qu'il parcourera l'arbre jusqu'à ce qu'il rencontre un noeud où une dérivation possible existe.

Voyons maintenant deux propriétés relatives à la fin de parcours du pointeur $->_{ob}$.

Propriété 1 : Soit $t \in T(\Sigma)^1_0$

$$\begin{array}{ccc} SO & * & SO' \\ | & |- & | \\ t & & <-_{ob} \\ & & | \\ & & u \end{array} \Rightarrow \begin{cases} t \xrightarrow{*} u \\ \text{et } u \in T(\Sigma)^1_0 \\ \text{et } u \text{ maximal pour } R \end{cases}$$

Démonstration :

- u maximal pour R :

Le pointeur $->_{ob}$ force la première dérivation possible rencontrée lors de son parcours et se transforme ensuite en $->_{ob}'$.

Si on a la configuration $\begin{array}{c} SO' \\ <-_{ob} \\ | \\ u \end{array}$, c'est donc qu'aucune dérivation de R n'est possible dans u.

Donc u est maximal pour R.

- $u \in T(\Sigma)^1_0$

Les pointeurs $->_{ob}$ et $->_{fac}$ permettent d'appliquer dans t des règles de R. Les règles gérant ces deux pointeurs ne modifient pas la structure des arbres où on les applique.

Donc :

$u \in T(\Sigma)^1_0$ et $t \xrightarrow[R]{*} u$

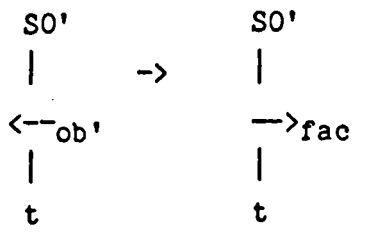
Propriété 2 : Soit $t \in T(\Sigma)^1_0$

$$\left.\begin{array}{ccc} SO' & * & SO' \\ | & |- & | \\ ->_{ob} & & <-_{ob}' \\ | & & | \\ t & & u \end{array}\right\} \Rightarrow \begin{cases} \exists\, (g \to d) \in R \text{ tel que} \\ \\ t \xrightarrow[g \to d]{} u \end{cases}$$

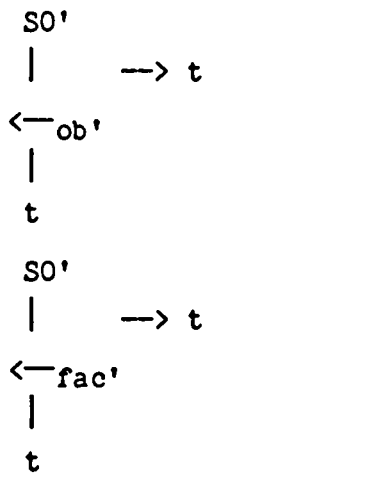
et aucun des arbres intermédiaires
ne contient le pointeur $\rightarrow_{fac}$

Démonstration : Puisqu'aucun arbre intermédiaire ne contient de pointeur $\rightarrow_{fac}$, le pointeur $<-_{ob'}$ du membre droit a été engendré par le pointeur $\rightarrow_{ob}$ du membre gauche.
Celui-ci a donc forcé une dérivation dans t. Or cette dérivation est en fait l'application d'une règle de R. Ce qui démontre la propriété.
Remarque : La proposition1 nous fournit une condition de maximalité exprimée sous forme de configuration d'arbre, donc exploitable par un système de réécriture.
Il faut, pour terminer la présentation des structures de contrôle, introduire les règles suivantes :

1. Pour relancer le pointeur $\rightarrow_{fac}$:

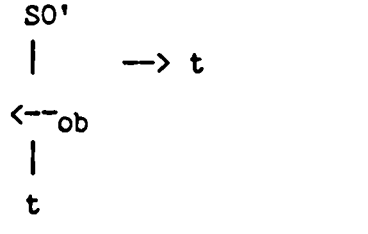


2. Pour arrêter les dérivations :



(ceci dans le cas où on s'autorise à arrêter le processus de dérivation avant d'avoir atteint un arbre maximal)

3. Pour terminer la réécriture :



Des rôles respectifs de $\longrightarrow_{ob}$ et $\dashrightarrow_{fac}$, on peut déduire les lemmes suivants :

Soit $t \in T(\Sigma)^1_0$, R système de réécriture linéaire sur Σ, R_{cont} le système contrôlé associé à R.

Lemme 1 : A toute suite de dérivation de R dans t, on peut faire correspondre une suite de dérivations de R_{cont} dans $\begin{matrix} S0 \\ | \\ t \end{matrix}$ telle que :

$$t \xrightarrow[R]{*} u \quad \Rightarrow \quad \begin{matrix} S0 \\ | \\ t \end{matrix} \xrightarrow[R_{cont}]{*} u$$

Démonstration :

Il suffit de n'utiliser que le pointeur $\dashrightarrow_{fac}$ pour effectuer, dans $\begin{matrix} S0 \\ | \\ t \end{matrix}$, les règles de R aux endroits et dans l'ordre où elles sont appliquées

dans $t \xrightarrow[R]{*} u$

Lemme 2 : Soit R linéaire défini sur $T(\Sigma)$, R_{cont} le système contrôlé associé.

Soient v, w deux termes d'une des formes suivantes :

$$\begin{matrix} S0 & S0' & S0' \\ | & | & | \\ t & \longrightarrow & \longleftarrow \\ & | & | \\ & t & t \end{matrix}$$

où $\longrightarrow$, $\longleftarrow$ sont des pointeurs de contrôle ($\longrightarrow_{fac}$, $\dashrightarrow_{ob}$ et leurs dérivés).

Plus exactement :

v de la forme $\begin{matrix} S0 \\ | \\ t \end{matrix}$ ou $\begin{matrix} S0' \\ | \\ \longrightarrow \\ | \\ t \end{matrix}$ $\quad t, u \in T(\Sigma)$

w de la forme $\overset{SO'}{|}\ \overset{SO'}{|}$

$\overset{|}{<--}$ ou $\overset{|}{-->}$

$\overset{|}{u}\ \overset{|}{u}$

alors à toute dérivation

$$v \xrightarrow[R_{cont}]{*} \omega$$

On peut associer une dérivation $t \xrightarrow[R]{*} u$

Démonstration :

Par construction de R_{cont}

Lemme 3 : Si R est acyclique alors R_{cont} est acyclique à partir de $\overset{SO}{|}\ t$,

$t \in T(\Sigma)$

Démonstration :

Les règles de contrôle qui complètent R en R_{cont} ne créent pas de cycles par construction. Les seuls cycles possibles sont donc induits par R.

Lemme 4 : Si R est noéthérien, R_{cont} est noéthérien à partir de $\overset{SO}{|}\ t$,

$t \in T(\Sigma)$.

Démonstration : A tout arbre $t \in T(\Sigma)$ et à toute règle de R applicable dans t sont associées des règles de R_{cont} en nombre fini.

Nous avons pu définir une structure de contrôle permettant de surveiller l'application des règles d'un système de réécriture linéaire à gauche, sans perdre les propriétés principales de celui-ci.

Elle nécessitait cependant un marquage au sommet de l'arbre à dériver. Cette condition, qui ne peut s'exprimer en termes de systèmes de réécriture, était indispensable et peu naturelle.

Le marqueur de sommet joue plusieurs rôles :

1. Générer un et un seul pointeur .

2. Donner une condition d'arrêt des dérivations (mise sous forme normale)

3. Servir de "butoir" aux pointeurs de simulation, comme nous le verrons plus loin.

La structure de contrôle que nous allons définir maintenant reprend l'idée précédente, mais permet de marquer des "sous-sommets", jouant les rôles du précédent marquage, de telle façon que cela peut être décrit par des systèmes de réécriture.

Nous allons créer des règles permettant d'isoler aléatoirement certains sous-arbres du terme à dériver, sous-arbres que l'on marquera au sommet ensuite.

Dans ces sous-arbres, on pourra alors utiliser la structure de contrôle précédente.

Plus précisément, nous opérerons de la manière suivante :

- Nous créons des règles permettant à chaque symbole d'engendrer un pointeur ($-->_{aux}$) et un seul
- Celui-ci pourra soit remonter dans l'arbre, soit se transformer en pointeur de gel ($-->_{gel}$) dont le rôle sera d'isoler le sous-arbre dont il est sommet
- Une fois qu'un sous-arbre est isolé, on peut utiliser à l'intérieur de celui-ci les structures de contrôle définies précédemment.

Les règles que nous créerons seront toutes linéaires.

La possibilité d'engendrer des pointeurs $-->_{aux}$ aléatoirement dans l'arbre, le non-déterminisme de ce pointeur permettront d'exhiber pour toute règle du système de départ R, une suite de dérivations la simulant de façon contrôlée.

I - Génération des pointeurs $-->_{aux}$:

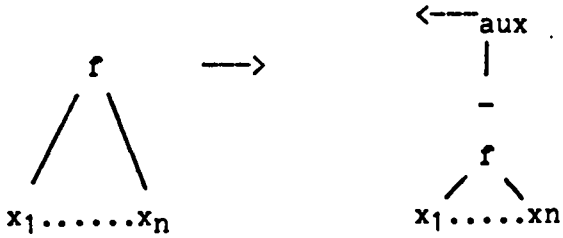
Soit Σ l'alphabet sur lequel agit R, système de réécriture que nous voulons simuler de façon contrôlée.

Soit $\Sigma = \{ \overline{f} / f \in \Sigma \}$

Afin d'éviter la créations de cycles et la présence de symboles parasites dans les sous-arbres que nous voulons isoler, nous autorisons pour

chaque noeud de l'arbre, la création d'un seul pointeur $\longrightarrow_{aux}$ en "barrant" le symbole du noeud correspondant :

$\forall f \in \Sigma$

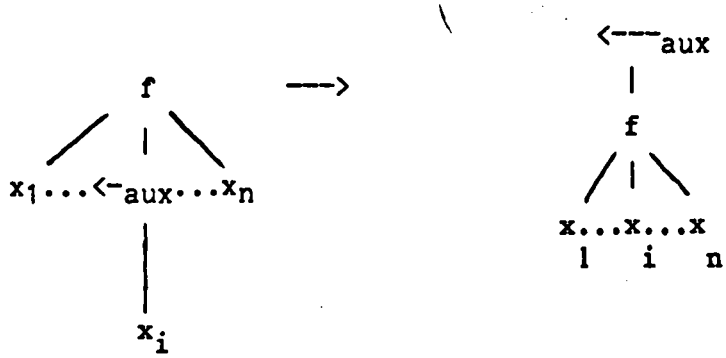


Permettons à ce pointeur :

1. de remonter :

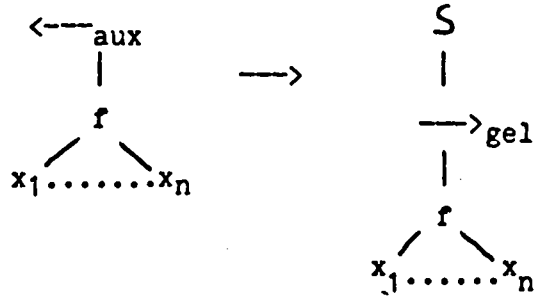
$\forall f \in \Sigma$

$\forall i \leq ar(f)$



2. de commencer le gel du sous-arbre

$\forall f \in \Sigma \cup \overline{\Sigma}$

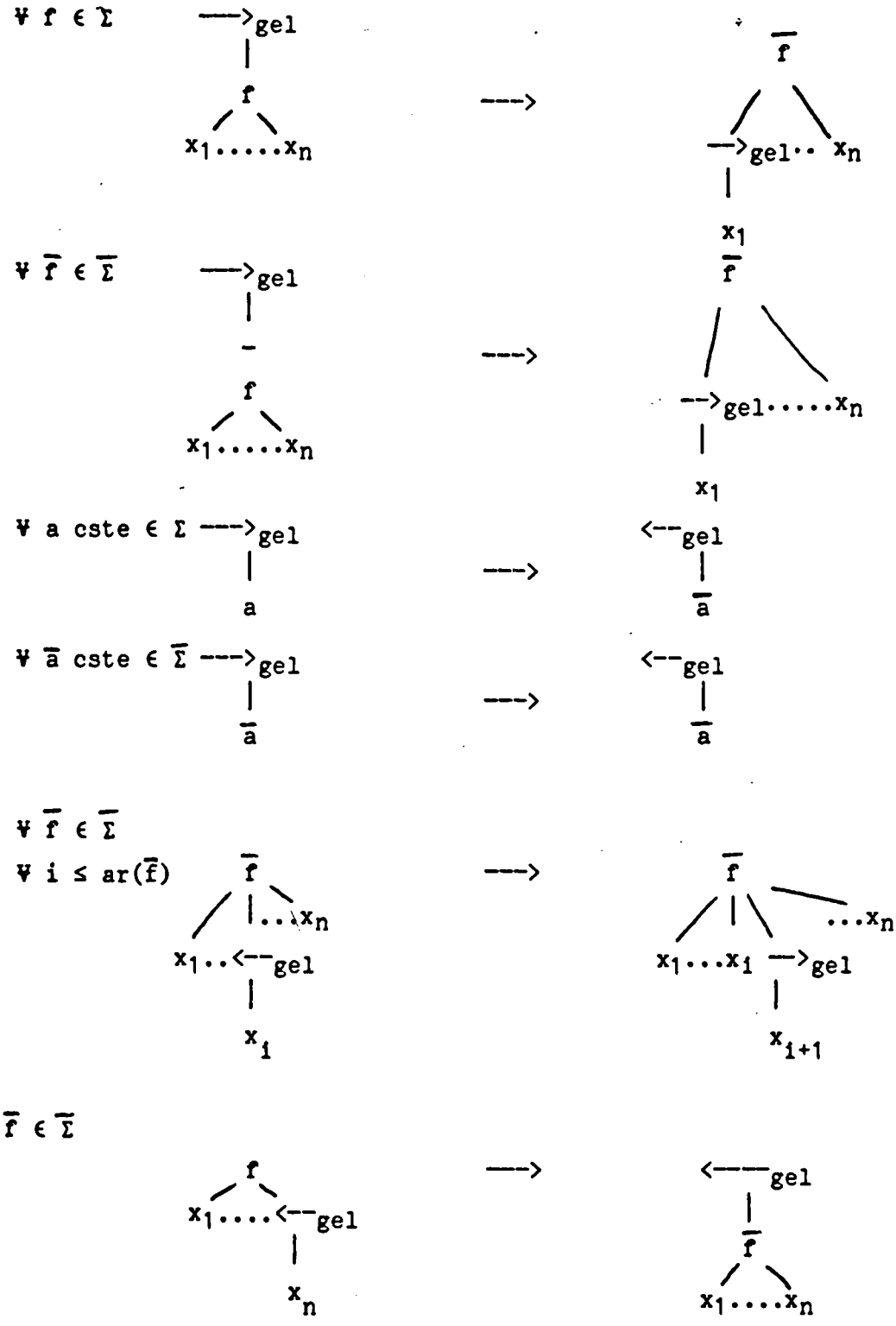


(1)

$\longrightarrow_{aux}$ ainsi défini peut donc choisir d'isoler un sous-arbre.

2. Gel d'un sous-arbre

Cette actions a pour but de "barrer" tous les noeuds du sous-arbre évitant ainsi la création de nouveaux pointeurs $<--_{aux}$, d'une part, d'autre part d'éliminer du sous-arbre tous les pointeurs en surnombre.



Remarque : Il n'est pas utile de définir de règles similaires aux deux dernières pour f ∈ Σ. En effet, étant donné l'ordre dans lequel $\longrightarrow_{gel}$ parcourt l'arbre, seuls les cas présentés peuvent se produire.
Cette première série de règles permet de "barrer" tous les noeuds de l'arbre. Voyons maintenant comment interpréter puis régler les différents cas où le pointeur $\longrightarrow_{gel}$ rencontre <---aux ou S.

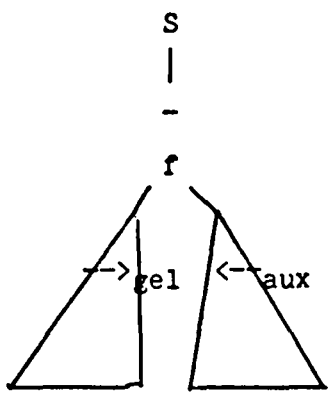
1. **Rencontre $\longrightarrow_{gel}$ (ou gel) et $\longleftarrow_{aux}$:**

Il y a à priori quatre cas possibles :

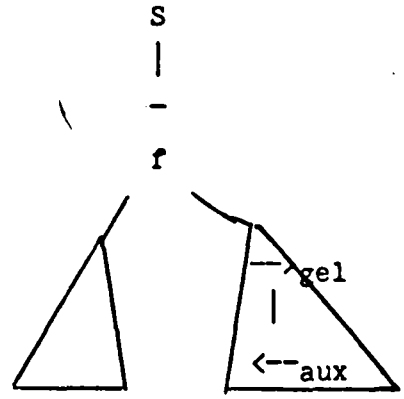
a. $\longrightarrow_{gel}$ correspondant aux schémas suivants

$\longleftarrow_{aux}$

x



1er temps $\longleftarrow_{aux}$ naît dans un sous-arbre non encore gelé



2ème temps : $\longrightarrow_{gel}$ continue son parcours. $\longleftarrow_{aux}$ ne pouvant remonter plus haut que f, on obtient la configuration

Nous choisissons de régler cette rencontre en créant la règle :

$\longrightarrow_{gel}$ | $\longleftarrow_{aux}$ | x $\longrightarrow$ $\longrightarrow_{gel}$ | x

dont nous dégagerons les propriétés plus loin.

b. $\longleftarrow_{gel}$ | $\longleftarrow_{aux}$ | x ne peut se présenter

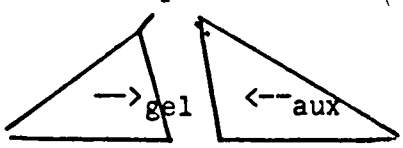
En effet, il faudrait que $\longleftarrow_{aux}$ soit né dans x après le passage de $\longrightarrow_{gel}$ dans x (sinon la règle précédente aurait été appliquée).
Or cela n'est pas possible, car tous les symboles de x ont été "barrés" lors du passage de $\longrightarrow_{gel}$.

c. $<\!\!-\!\!-_{aux}$
|
$-\!\!>_{gel}$ ne peut se présenter non plus
|
x

En effet cette configuration implique qu'on ait eu auparavant la configuration $-\!\!-\!\!>_{gel}$
|
$<\!\!-\!\!-_{aux}$ (ces deux pointeurs parcourent l'arbre par sauts d'un niveau au plus chaque fois)
|
x

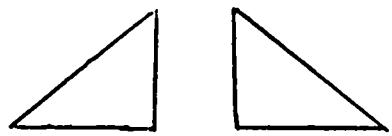
Or il n'existe aucune règle permettant cette transformation.

d. $<\!\!-\!\!-\!\!-_{aux}$
|
$<\!\!-\!\!-\!\!-_{gel}$ le seul cas où cette configuration peut se présenter (l'autre étant réglé par a) correspondrait aux schémas suivants :
|
x
S
|
-
f



1er temps : $<\!\!-\!\!-_{aux}$ naît dans une région non encore barrée

S
|
$<\!\!-\!\!-\!\!-_{aux}$
$<\!\!-\!\!-\!\!-_{gel}$
f



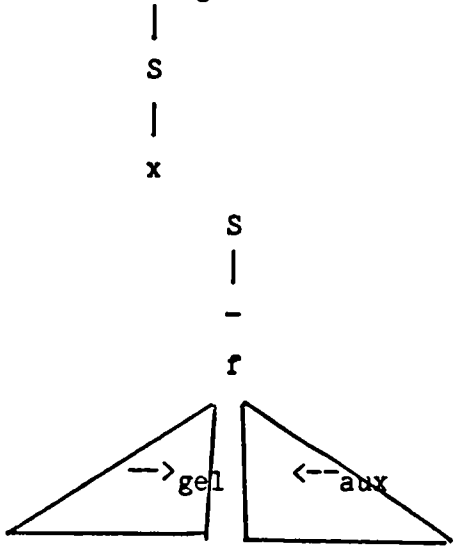
2ème temps $<\!\!-\!\!-_{aux}$ remonte sous S
$<\!\!-\!\!-_{gel}$ achève son parcours

Or comme nous le remarquions précédemment, $<\!\!-\!\!-_{aux}$ ne peut remonter les symboles barrés.

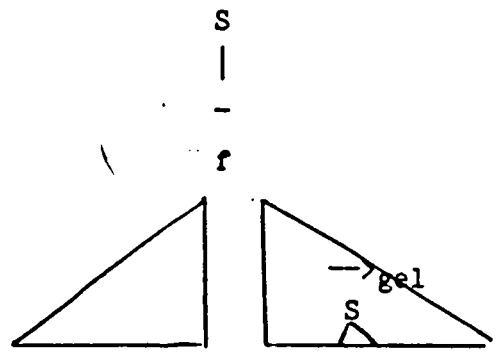
Cette configuration ne peut donc se présenter.

2. **Rencontre de $\longrightarrow_{gel}$ (ou $\longleftarrow_{gel}$) et de S :**

a. $\longrightarrow_{gel}$ correspond aux schémas suivants :



1er temps : un pointeur $\longleftarrow_{aux}$ nait dans une région non barrée

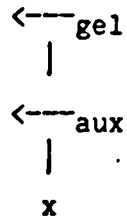


2ème temps : $\longleftarrow_{aux}$ devient S
$\longrightarrow_{gel}$ continue son parcours.

Nous choisirons de régler ce cas en privilégiant l'opération de gel la plus interne. En clair, cela signifie que nous ne donnerons pas de règles de réécriture gérant cette rencontre. $\longrightarrow_{gel}$ sera donc bloqué au-dessus de S tant que les dérivations dans le sous-arbre interne ne seront pas terminées.

Lorsqu'elles seront terminées, une règle permettra, nous le verrons, d'effacer le marqueur de sommet, et $\longrightarrow_{gel}$ pourra achever son parcours.

b. $\longleftarrow_{gel}$ pour obtenir cette configuration, il faudra avoir eu précédemment



qui ne peut se présenter

Cette configuration ne peut donc pas non plus se présenter.

c. S
|
se présente - au début de la phase de gel
-->gel - si l'on a eu précédemment <---aux
| |
x , ce qui est
---->gel
impossible |
x

S et $--->_{gel}$ sont donc deux symboles associés, leur rencontre n'est pas un cas particulier.

d. S
|
se présente - lorsque $--->_{gel}$ a parcouru entièrement x
<--gel - après la configuration <---aux
| |
x qui ne peut
<---gel
|
se présenter x

Lorsqu'on obtient cette configuration, cela signifie donc que $--->_{gel}$ est revenu à son point de départ.

3. **Rencontre de $\longrightarrow_{gel}$(ou $<\!\!---_{gel}$) et $\longrightarrow_{gel}$(ou $<\!\!---_{gel}$)**

(P) Aucun de ces quatre cas de rencontre ne peut se produire, car :

- Chaque S est associé à un seul pointeur $--->_{gel}$
- Tout pointeur de gel reste entre le S qu'il a créé et le premier S du sous-arbre à geler.

Les pointeurs de gel sont donc seuls dans ces parties d'arbres.

Ceci nous permet donc d'énoncer :

<u>Proposition</u> :

La configuration S signifie exactement que $t \in T(\bar{\Sigma})$
|
<--gel
|
t

(ou encore, tous les noeuds de t sont "barrés")

Démonstration :

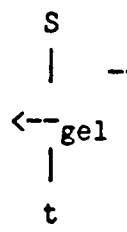
a) t ne contient pas de symbole S (sinon $<---_{gel}$ ne serait pas remonté)
t ne contient ni $--->_{gel}$, ni $<---_{gel}$ (d'après (P)).
t ne contient pas de $<---_{aux}$ ($--->_{gel}$ les a tous rencontrés et éliminés)

Donc t ne contient que des symboles de $\Sigma \cup \overline{\Sigma}$.

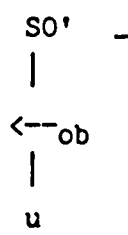
b) $--->_{gel}$ ayant parcouru exhautivement t, y barrant tous les symboles de Σ qu'il a rencontré.

Donc $t \in T(\overline{\Sigma})$.

Grâce à ce résultat, nous pouvons terminer la définition de cette structure de contrôle, à l'aide des règles.

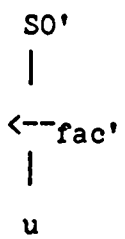


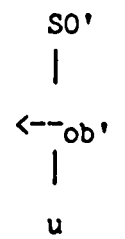
---> SO permettant d'appliquer la première structure de contrôle



—→ u permettant aux éventuels $--->_{gel}$ immobilisés aux niveaux supérieurs de continuer leur parcours lorsque la simulation est terminée

On peut créer aussi les règles suivantes, si l'on s'autorise à restituer un sous-arbre à son contexte dès qu'une dérivation a été effectuée :





Remarques :

1. Lorsqu'on utilise la première structure de contrôle, il ne faut pas oublier de le définir sur $\overline{\Sigma}$ et non plus sur Σ.

2. En appliquant cette structure de contrôle à $t \in T(\Sigma)$, on obtient après plusieurs dérivations $t_1 \in T(\Sigma \cup \overline{\Sigma})$.

Pour éviter cela, il serait judicieux, avant d'effacer les marqueurs de sommet, de créer un pointeur qui renvoie les lettres "barrées" dans Σ, et de n'effacer le marqueur de sommet que lorsque le sous-arbre appartient à $T(\Sigma)$.

Nous supposerons par la suite que le système de réécriture effectuant ce travail existe, mais ne le détaillerons pas.

Examinons maintenant les propriétés de cette structure de contrôle :

Soit R un système de réécriture linéarisé à gauche sur $T(\Sigma)$

R_{cont} le système contrôlé correspondant (pour la structure de contrôle que nous venons de définir).

Théorème 1 : Soient $t, u \in T(\Sigma)$

A toute dérivation $t \xrightarrow[R]{*} u$ on peut faire correspondre $t \xrightarrow[R_{cont}]{*} u$

Démonstration :

Il suffit d'engendrer des pointeurs $\xleftarrow{\quad}_{aux}$ aux occurences de t où doivent s'appliquer les règles de R, dans l'ordre où celles-ci doivent s'appliquer, puis de geler les sous-arbres correspondants pour appliquer, grâce au pointeur $\longrightarrow_{fac}$, la règle au sommet du sous-arbre.

L'effacement des "barres" nous restituera u à la fin de la dérivation.

Theorème 2 : Soient $u, t \in T(\Sigma)$

A toute dérivation $t \xrightarrow[R_{cont}]{*} u$ on peut associer une dérivation $t \xrightarrow[R]{*} u$

Démonstration :

Lors de la dérivation $t \xrightarrow[R_{cont}]{*} u$, un certain nombre de sous-arbres ont été barrés (entièrement) puis "débarrés".

(Ce sont les seules suites de règles initialement applicables)

Si aucun sous-arbre n'a été isolé de son contexte alors $t = u$ et la propriété est vérifiée.

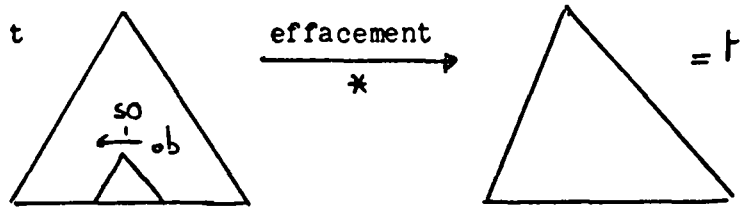
Sinon c'est qu'une règle du premier système de contrôle a été utilisée et d'après les résultats qui s'y rapportent, au moins une règle de R a été appliquée (et seules des règles de R ont été appliquées).

Remarque 1 . Pour cette démonstration nous avons utilisé :

- Le fait qu'il existe un système de réécriture effaçant les barres.

2 . Cette structure de contrôle a l'avantage sur la première décrite qu'elle s'exprime entièrement en termes de systèmes de réécriture. Elle possède pourtant l'inconvénient majeur d'induire des cycles.
En effet, si t est sous-forme normale pour R on pourra :

- Isoler un sous-arbre de t
- Essayer, grâce à la première structure de contrôle de dériver ce sous-arbre (on obtiendra : t



ce qui forme un cycle.
Nous n'utiliserons donc cette méthode que dans la partie consacrée aux théories équationnelles, où la notion de terminaison finie a moins d'importance que pour les systèmes de réécriture, pour lesquels nous utiliserons la première structure de contrôle.

CHAPITRE III

Linéarisation à droite d'une règle de réécriture

Linéarisation à droite d'une règle de réécriture

Introduction :

La linéarisation d'une règle de réécriture peut se décomposer en trois parties :

- linéarisation à droite
- linéarisation à gauche
- lien entre les deux premiers points.

La première phase, qui fait l'objet de ce chapitre, consistera à recopier dans le membre droit les sous-arbres instanciant des variables occurant plusieurs fois dans ce terme.
Elle commencera à s'effectuer lorsque l'une quelconque des deux structures de contrôle aura engendré un pointeur $\longrightarrow_{fac}$.
Afin de contrôler à tout instant les dérivations à accomplir, nous définirons plusieurs familles de règles de réécriture (regroupées en "primitives") s'enchaînant dans un ordre précis.
Cela sera possible grâce aux différents "pointeurs" dont la présence sera nécessaire pour qu'une dérivation puisse être appliquée. D'autre part, un seul pointeur sera présent à la fois dans un sous-arbre isolé de son contexte (i.e : en cours de simulation d'une règle du système de départ), assurant un contrôle total du travail en cours.

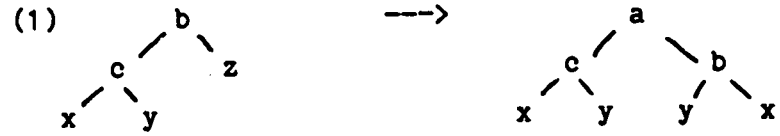
Pour effectuer la linéarisation à droite d'une règle de réécriture, nous modifions le membre droit de la règle à simuler en y introduisant de nouveaux symboles fonctionnels ayant pour rôle d'indiquer qu'une copie est à faire, l'endroit où elle doit se faire, ainsi que le noeud où commence le modèle.

Rappelons enfin que la méthode décrite ne s'applique qu'à des termes clos et ceci afin de pouvoir définir les règles permettant aux différents pointeurs de parcourir les termes.

II. Mode opératoire :

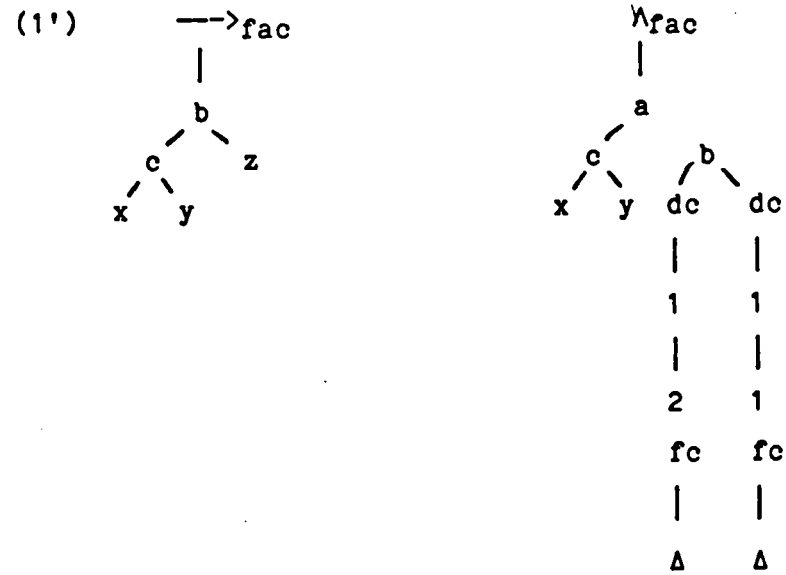
Nous l'illustrons sur un exemple :

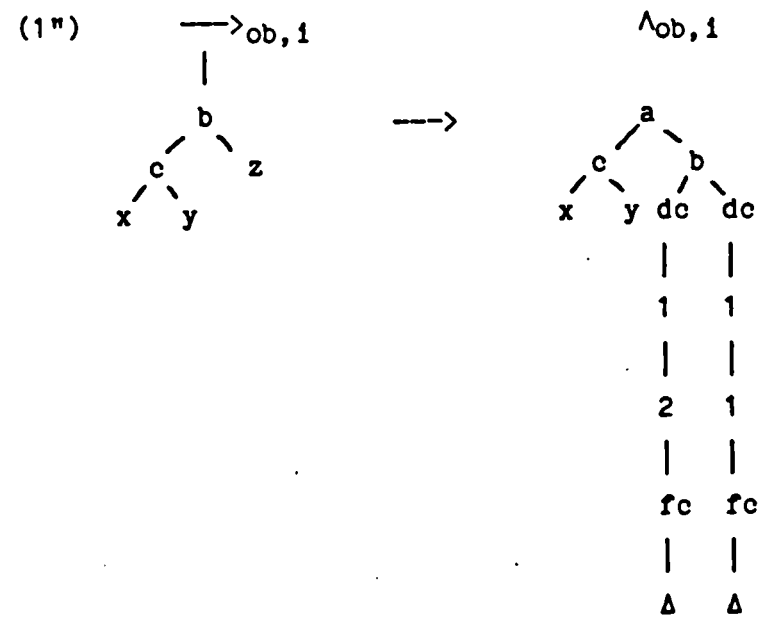
Soit la règle non linéaire à droite :

(1) 

Nous allons la décomposer en une suite de règles permettant d'aller recopier aux noeuds adéquats les valeurs des variables correspondantes.

Tout d'abord, nous modifions cette règles de la manière suivante :

(1') 

(1") 

Si la règle de départ est la ième règle de sommet de membre gauche égal à b.

Λ_{fac}, $\Lambda_{ob,i}$ sont des marqueurs de sommet qui serviront (nous le verrons plus loin) :

a) A regénérer les pointeurs $\longrightarrow_{fac}$ ou $\longrightarrow_{ob}$ (ou leurs dérivés) en fin de travail

b) De point de repère pour les différents pointeurs parcourant le sous-arbre.

- Δ est une nouvelle constante repérant l'endroit où doit s'effectuer la prochaine copie d'un symbole.
- Les suites d^c , $(\alpha_i) \in N$ indiquent que le sommet du sous-arbre à

$$d^c$$
$$|$$
$$(\alpha_i)$$
$$|$$
$$fc$$

recopier en Δ se trouve à l'occurence $\overline{\alpha_1 \ldots \alpha_n}$).

A partir de ce membre droit désormais linéaire, on peut alors définir le travail de copie à accomplir sous la forme d'un algorithme, dont chaque action et prédicat s'exprimera en termes de systèmes de réécriture et de configuration d'arbre.

Linéarisation à droite

début
Engendrer un pointeur de recherche ;
chercher un noeud où doit s'effectuer une copie ;
tant que non fini1 faire
% Recopier un sous-arbre %
Remonter le code (α_i) au sommet ;

Descendre au noeud de code $\alpha_1 \ldots \alpha_n$;
Tant que non fini 2 faire
% Recopier un symbole %
Chercher le symbole à recopier ;
Charger l'information ;
Remonter l'information au sommet ;
Descendre jusqu'au premier Δ ;
Déposer l'information et remonter sous le sommet ;

Fait

Effacer les symboles auxiliaires ;

Regénérer un pointeur de recherche ;

Fait

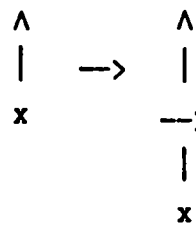
Effacer les symboles auxiliaires et regénérer le pointeur de dérivation ;

Fin.

Nous allons maintenant décrire chacune de ces primitives :

1. Engendrer un pointeur de recherche

On exprime cette action par :

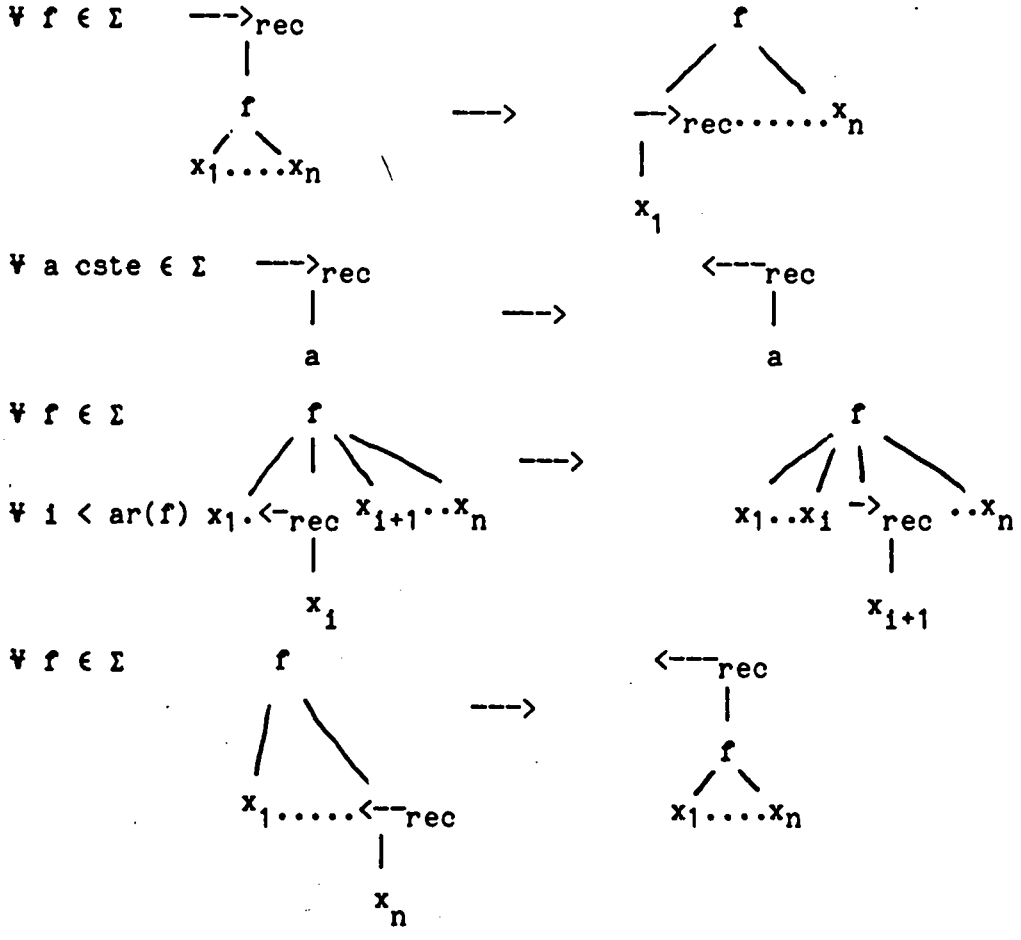


où Λ et Λ' correspondent au même pointeur de dérivation

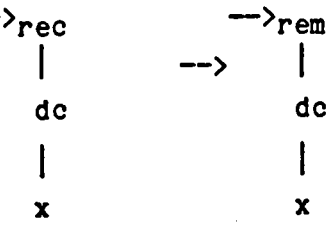
2. Chercher un noeud où doit s'effectuer une copie :

Nous munissons le pointeur de recherche $\dashrightarrow_{rec}$ de règles lui permettant d'explorer l'arbre conformément à l'ordre "sens de parcours", ceci afin qu'il repère le premier noeud (relativement à cet ordre) où doit s'effectuer une copie. En clair, il devra repérer le premier symbole dc.

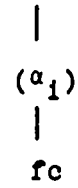
C'est ce qu'exprime le système de réécriture suivant :



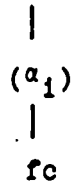
Ces quatre familles de règles gèrent le parcours du pointeur $\longrightarrow_{rec}$. La règle suivante exprime sa transformation en pointeur de remontée lorsqu'il a rencontré dc :



3. Remonter la suite dc sous le sommet Λ'



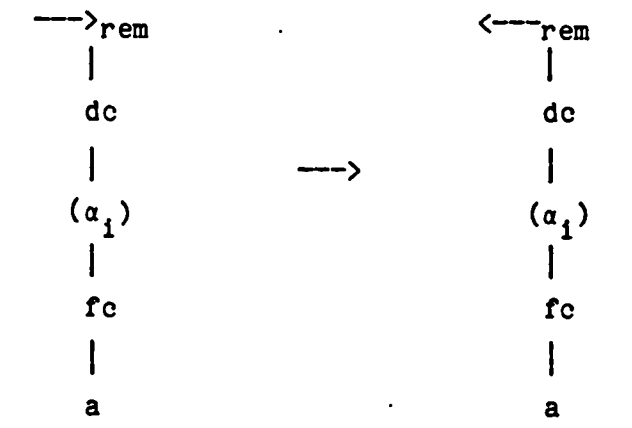
Afin d'obtenir des règles de parcours homogènes et de pouvoir considérer, dans le chapitre traitant des théories équationnelles, la reversibilité des règles, cette ascension de la suite



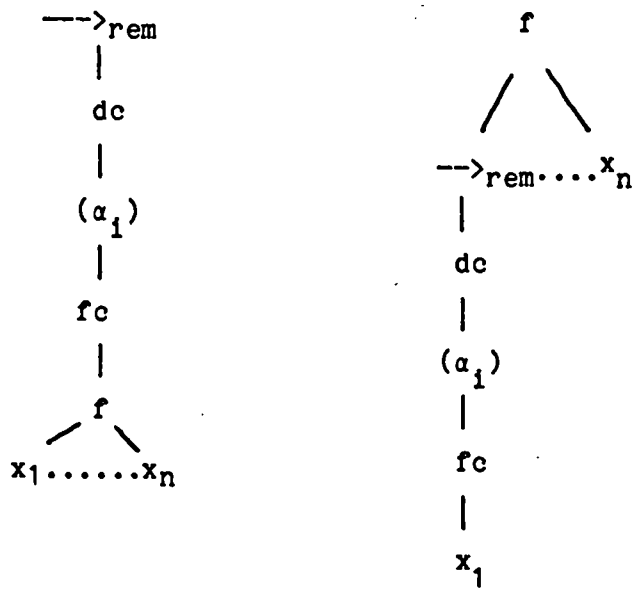
sera accomplie en lui faisant parcourir le reste de l'arbre conformément à l'ordre sens de parcours.

Ce qui nous conduit à écrire le système de réécriture suivant :

$\forall$ a constante $\in \Sigma \cup \{\Delta\}$

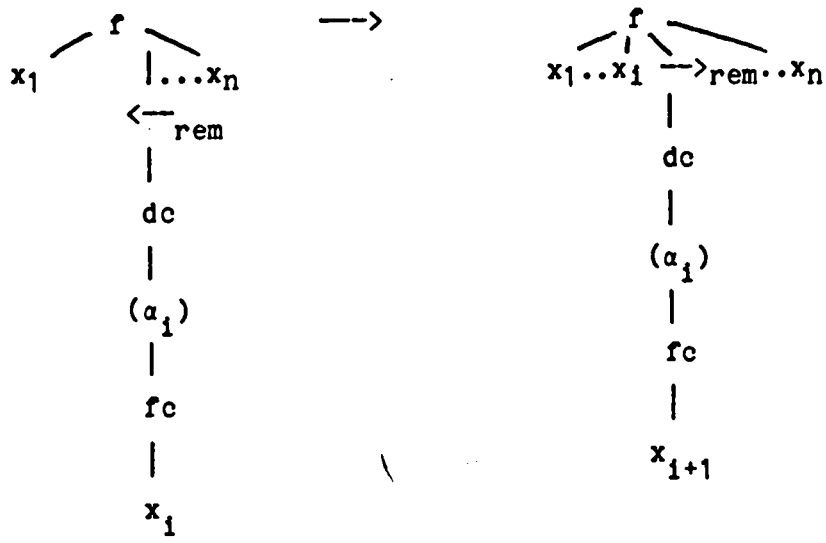


$\forall$ f $\in \Sigma$

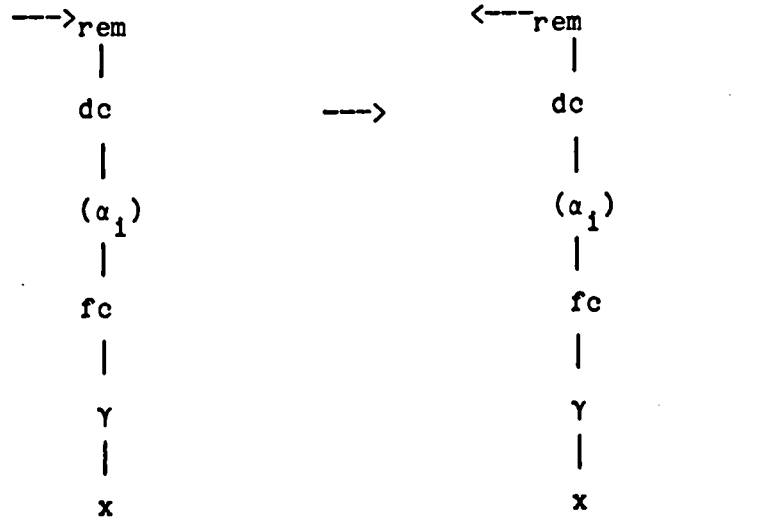


$\forall f \in \Sigma$

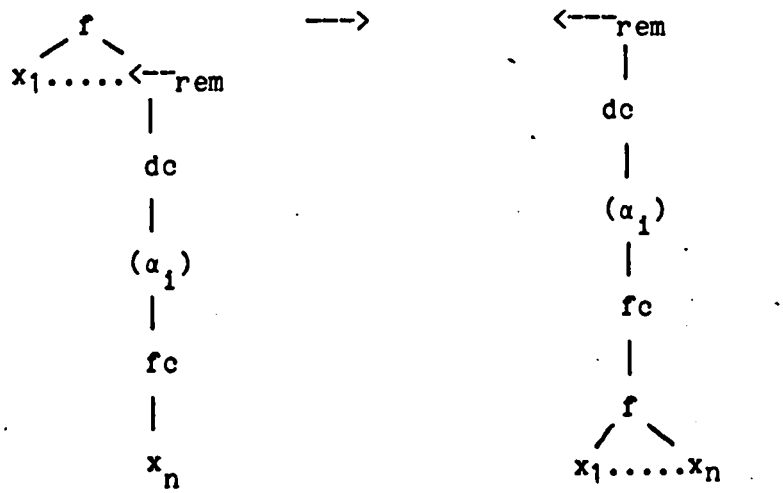
$\forall i < ar(f)$



$\forall \gamma \notin \Sigma$



$\forall f \in \Sigma$

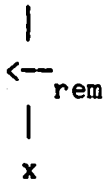


Remarque :

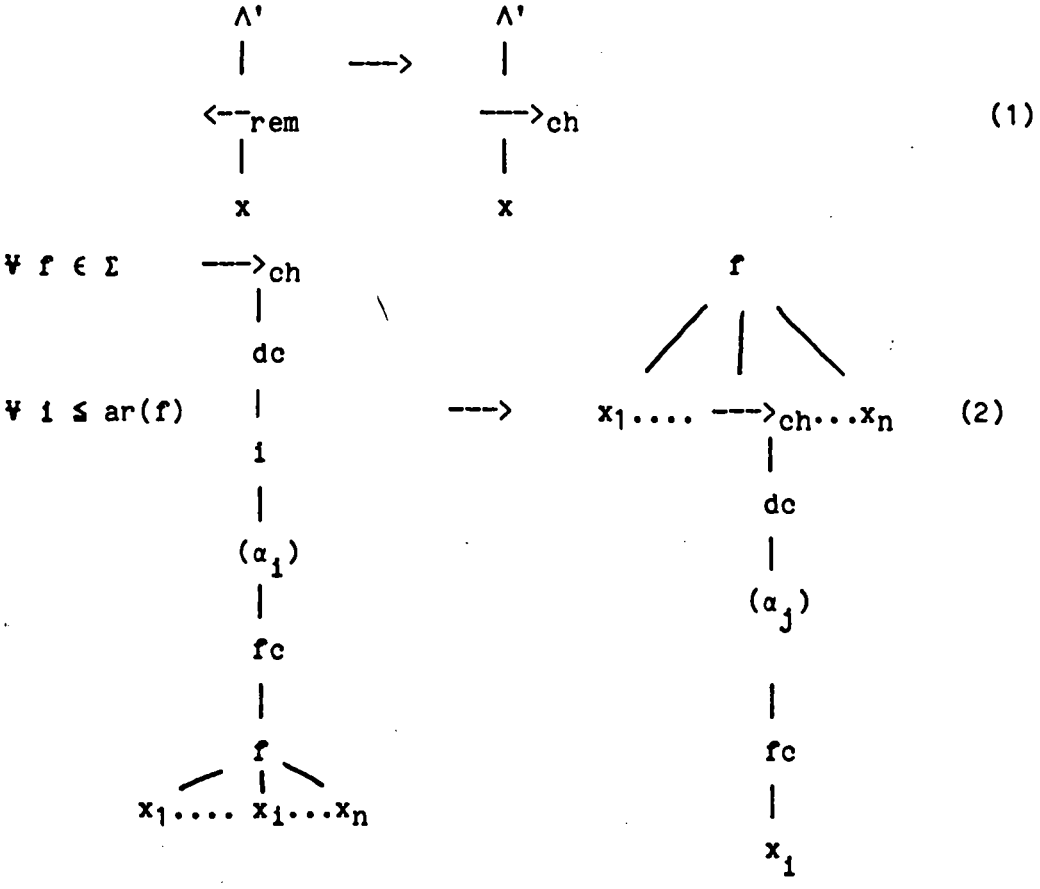
Les symboles $\gamma \notin \Sigma$ ont été, d'une part, définis comme étant d'arité 1 : ce sera toujours le cas (ce sont des informations insérées dans l'arbre originel : elles ne modifient pas la structure de celui-ci).

D'autre part, on les a traité comme des constantes de Σ : ils sont unaires, et, dans cette partie de la simulation, touts leurs descendants le sont aussi. Il n'est donc pas nécessaire de parcourir ceux-ci.

L'application de ce système de réécriture prendra fin lorsqu'on obtiendra la configuration : Λ'

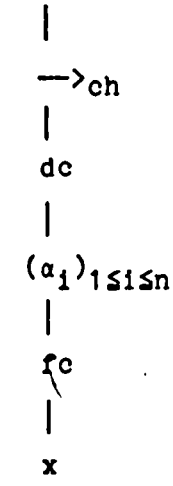


Nous pouvons alors décrire le système de réécriture permettant de descendre le pointeur jusqu'au noeud de code $\overline{\alpha_1 \ldots \alpha_n}$:

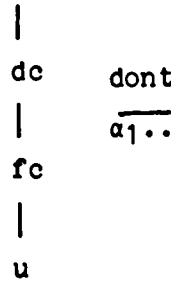


Proposition :

Si l'on part de la configuration A = Λ' où x ∈ T(Σ)



et si l'on applique les règles du système de réécriture décrites page précédente, on obtient la configuration —>ch , où u est un sous arbre de x

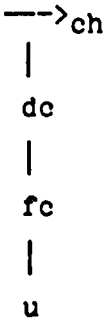


dont le sommet est de code $\overline{\alpha_1 \ldots \alpha_n}$ dans x.

Démonstration :

- Depuis la configuration A, on ne peut appliquer que des règles de la famille (2). Or celles-ci font descendre le pointeur d'un niveau dans x à chaque application.

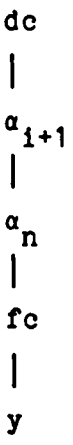
Puisque l'on part de la profondeur 0 (sommet de x), on descendra donc à la profondeur n dans x, où l'on arrivera lorsqu'on aura la configuration



Montrons que le sommet de u (qui est à la bonne profondeur) est bien celui correspondant au code $\overline{\alpha_1 \ldots \alpha_n}$.

Pour cela démontrons P1.

P1 : Soit une suite $(\alpha_1, \ldots \alpha_n)$. Si l'on est parti de la configuration A, et si l'on a la configuration B = $\xrightarrow[ch]{}$ dans x, alors le sommet de y est de code $\overline{\alpha_1 \ldots \alpha_i}$ dans x.



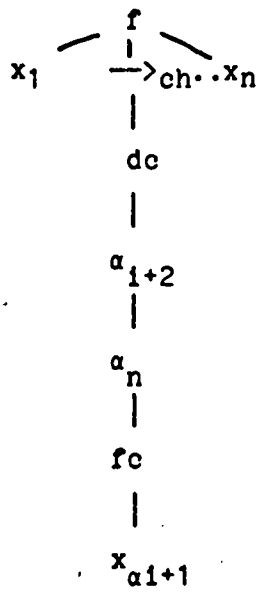
Démonstration :

P1 est vraie lorsqu'on a la configuration A

Supposons que P_1 soit vraie pour $0<i<n$

C'est dire que l'on a la configuration B, où y = $f(x_1 \ldots x_n)$ $f \in \Sigma$

On applique donc une règle de la famille (2) ce qui nous amène à la configuration suivante :



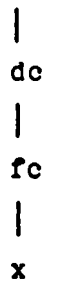
où le sommet de $x_{\alpha i+1}$ est de code $\overline{\alpha_1 \ldots \alpha_i . \alpha i+1}$ dans x.

P_1 est donc toujours vraie.

Ceci termine la preuve de P_1 et celle de la proposition (lorsque i = n).

Corollaire :

On arrive donc au noeud $\overline{\alpha_1 \ldots \alpha_n}$ relativement au sommet lorsque l'on obtient la configuration $\longrightarrow_{ch}$



Cette condition d'arrêt nous permet d'enclencher maintenant la phase de copie proprement dite.

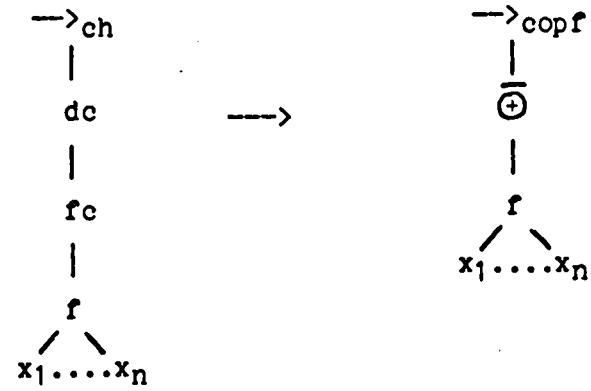
Le rôle de la primitive suivante sera :

1) de marquer le sommet du sous-arbre à recopier

2) de charger le symbole sommet de ce sous-arbre

Ce que nous décrivons par la famille de règles suivante :

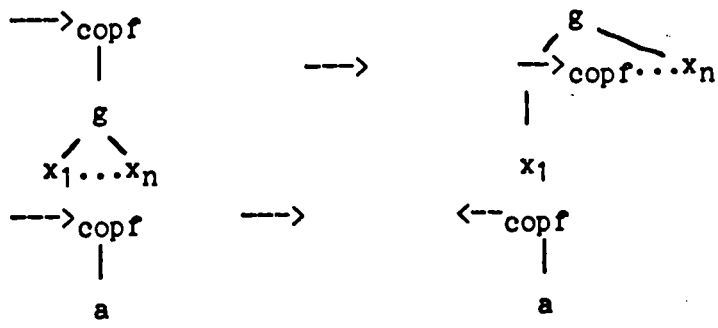
$\forall f \in \Sigma$



où $\overline{\oplus} \notin \Sigma$ est un nouveau symbole fonctionnel marquant la tête du sous-arbre à recopier.

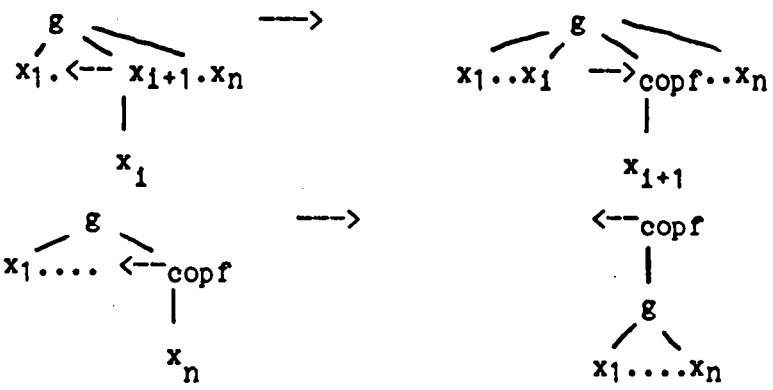
Définissons maintenant le système de réécriture permettant à $\longrightarrow_{copf}$ de rejoindre le sommet Λ' :

$\forall g \in \Sigma', \forall f \in \Sigma$



$\forall a$ cste $\in \Sigma', \forall f \in \Sigma$

$\forall g \in \Sigma' \; \forall f \in \Sigma$
$\forall i < ar(g)$



$\forall g \in \Sigma \setminus \{\Lambda'_{fac}, \Lambda'_{ob}\}$
$\forall f \in \Sigma$

Après application de cette primitive, un pointeur portant la valeur du sommet du sous-arbre à recopier se trouve sous le marqueur de sommet Λ'.
Il faut maintenant définir le système de réécriture qui permettra de déposer cette information à l'endroit adéquat.
On remarque facilement que l'information est à déposer sur le premier Δ relativement à l'ordre "sens de parcours" (c'est le Δ dont on a remonté le code (α_i) au sommet des phases précédentes).
La primitive [première copie] aura donc pour tâche :

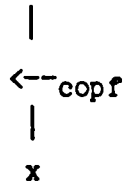
a. de rechercher dans l'arbre le premier symbole Δ

b. d'y déposer l'information portée par le pointeur $\longrightarrow_{copf}$

Ce que nous exprimons par le système de réécriture suivant, que l'on peut décomposer en trois parties :

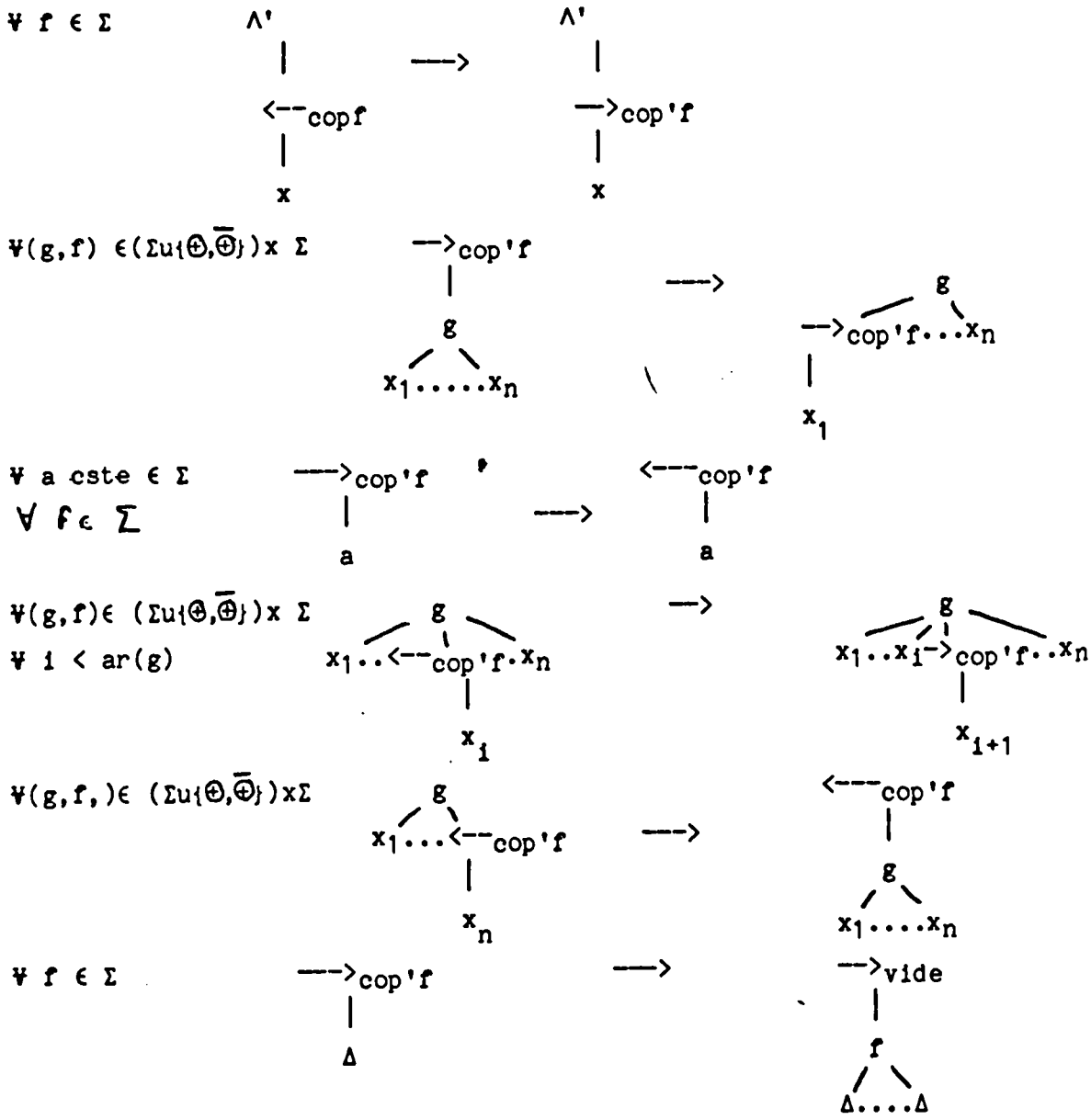
<u>première partie</u> : Remontée du pointeur $\longrightarrow_{copf}$ au sommet.

Cette primitive est associée au système de réécriture défini, ci-dessus, dont le travail se termine lorsque l'on obtient la configuration



(en effet, $\longrightarrow_{copf}$ parcourt exhaustivement l'arbre dans le sens de parcours depuis son lieu de naissance (descendant de Λ') jusqu'à Λ'(où il est forcément en phase ascendante). Lorsqu'il est sous Λ' en phase ascendante, il n'existe pas de règle dans le système de réécriture lui permettant de changer de place ou d'état).

deuxième partie : Recherche du premier Δ



Remarque :

Cette dernière familles de règles est elle aussi linéaire à droite car Δ est une constante, et non une variable.

A l'aide des primitives, il nous a été possible de recopier à l'endroit voulu le symbole sommet du sous-arbre modèle.

Nous devons maintenant définir d'autres primitives qui nous permettrons de recopier où il faut les autres symboles fonctionnels du sous-arbre modèle.

Ces primitives seront, dans leur ordre d'utilisation après la phase de première copie :

1) Remonter au sommet le pointeur vide

2) Chercher le sommet marqué d'un $\overline{\oplus}$ du sous-arbre modèle

3) Chercher dans ce sous-arbre le premier noeud (relativement à l'ordre sens de parcours) non encore recopié.

4) Charger la valeur de ce noeud et remonter au sommet

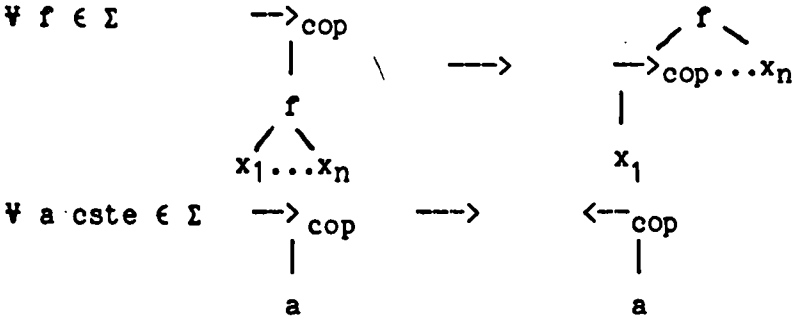
5) Aller déposer l'information au noeud adéquat.

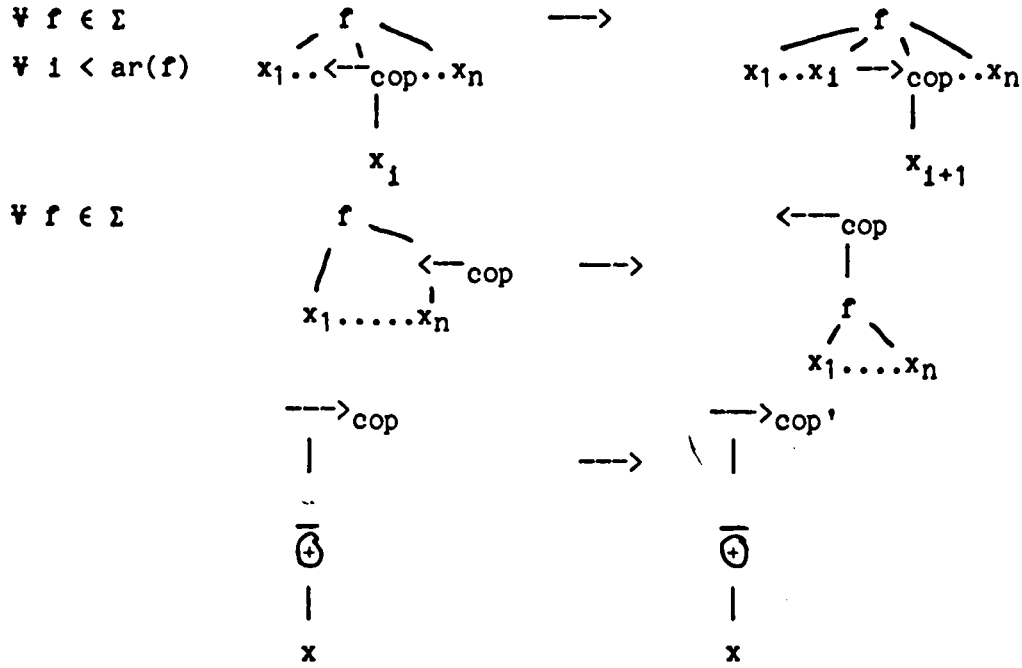
Lorsque les 4 premières phases auront été décrites en termes de systèmes de réécriture, nous montrerons que l'information devra être déposée sur le premier Λ de l'arbre, relativement à l'ordre sens de parcours.

1. Le système de réécriture gérant la remontée du pointeur $\longrightarrow_{vide}$ est du même type que les autres systèmes de remontée. La dernière règle en sera :

2. Chercher le sommet du sous-arbre à recopier

Ce travail va être accompli par le système de réécriture suivant :



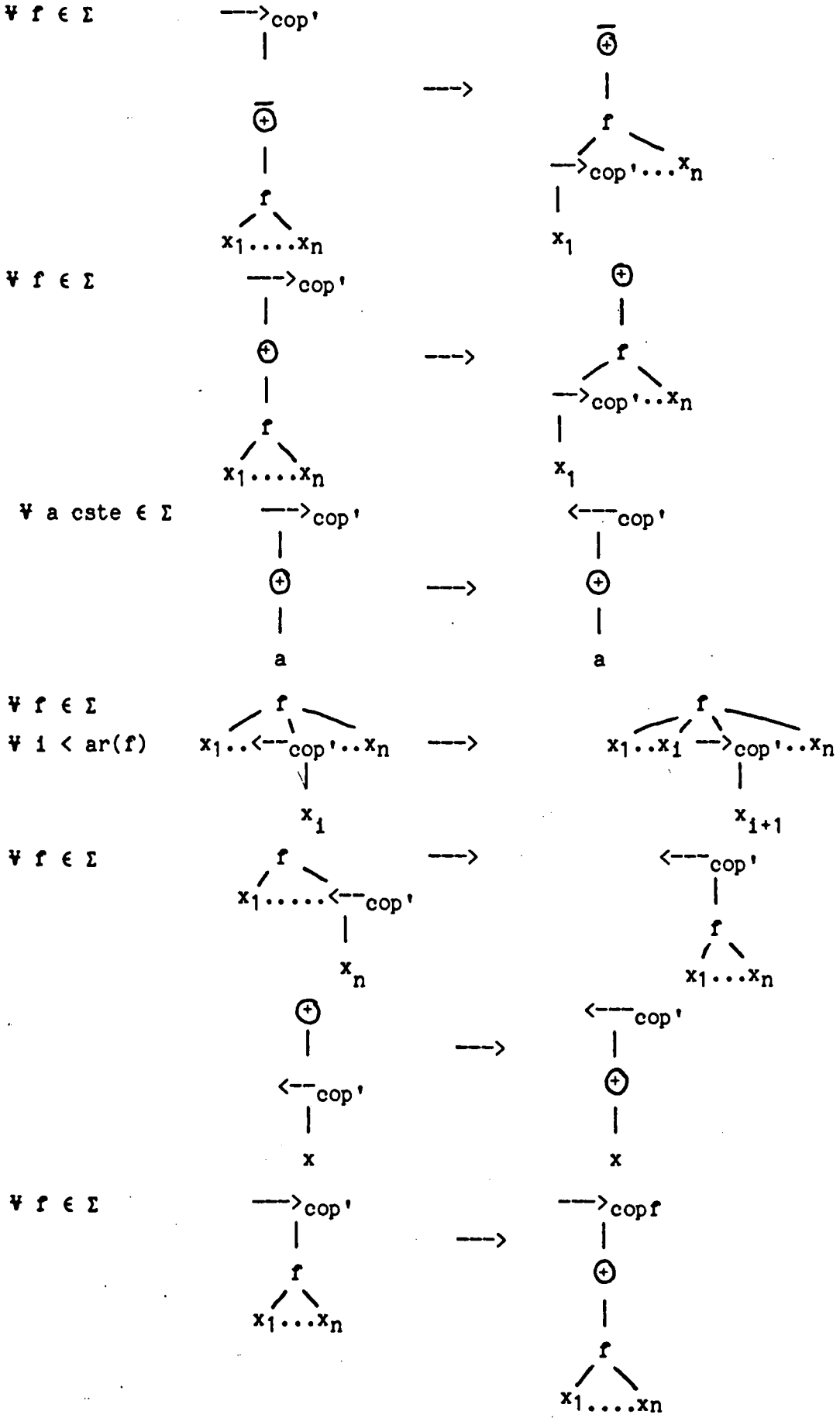


Ces règles, typiques d'une procédure de recherche relativement à l'ordre sens de parcours, nous permettent de repérer un symbole $\overline{\oplus}$.
(Nous montrerons plus loin, lors de la preuve de l'algorithme, que si l'on est parti d'un arbre appartenant à T(Σ), alors il y a au plus un symbole $\overline{\oplus}$ dans l'arbre).
Passons maintenant à la troisième phase de copie.
Précisons notre mode opératoire :

- tout noeud dont l'information a été chargée est marqué, lors de ce chargement, d'un $\oplus$.
- on charge les informations dans l'ordre sens de parcours

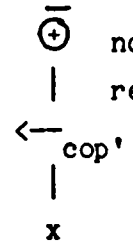
La recherche du noeud dont il faut charger la valeur consistera donc à parcourir le sous-arbre modèle, suivant l'ordre sens de parcours, jusqu'à ce que l'on trouve un noeud de ce sous-arbre non marqué d'un $\oplus$.
Remarque : Cette méthode fournit de plus une condition d'arrêt de la copie. En effet, le pointeur chargé de trouver le premier noeud non marqué d'un $\oplus$ ne reviendra au sommet du sous-arbre modèle que si tous les noeuds sont marqués, et donc recopiés (si l'on se réfère à la succession des différentes phases).
Décrivons donc cette phase :

Troisième partie :



Ce système vérifie les deux propriétés suivantes, que nous allons démontrer

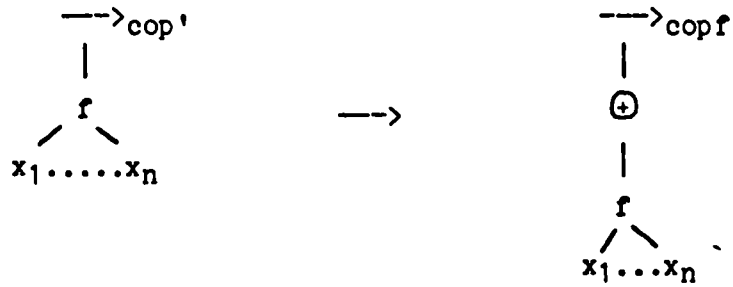
Propriété 1 : Le noeud dont l'information est chargée lors de l'application de ce système de réécriture est le premier du sous-arbre modèle non marqué d'un $\oplus$ ou d'un $\overline{\oplus}$

Propriété 2 : La configuration  nous assure que tout le modèle a été recopié

Démonstration de la propriété 1 :

. $\longrightarrow_{cop'}$ ne peut être le père d'un noeud dont le symbole appartient à $T(\Sigma)$ que si ce noeud n'est pas marqué. (Il ne pointe jamais, lorsqu'il est en phase descendante, sur un symbole marqué, mais sur le marquage lui-même) En effet lorsqu'il rencontre un $\oplus$ ou un $\overline{\oplus}$, le pointeur "descend deux étages", donc ne pointe pas sur le noeud marqué du $\oplus$ ou du $\overline{\oplus}$. (Il suffit de voir les deux familles de règles gérant cette rencontre).

. Si, en phase descendante, il pointe sur un symbole de $T(\Sigma)$, la seule règle applicable est de la famille



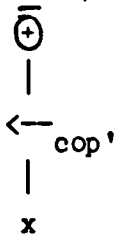
Donc

a. $\longrightarrow_{cop'}$ charge l'information d'un noeud non marqué d'un $\oplus$ ou d'un $\overline{\oplus}$ (point 1)

b. Ce noeud dont il charge l'information est le premier noeud non marqué (relativement à l'ordre sens de parcours) qu'il rencontre. (point 2).

Ce qui démontre la propriété 1.

Démonstration de la propriété 2 :

. Lorsque l'on a la configuration 

, cela signifie que $\longrightarrow_{cop'}$

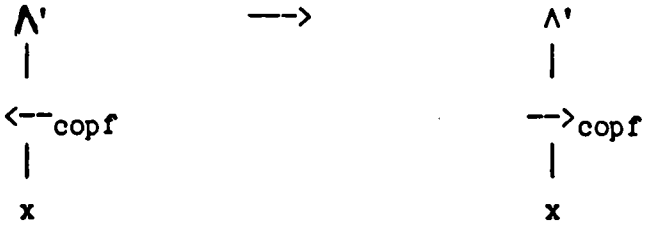
a parcouru exhaustivement x. S'il y existait un symbole non marqué d'un ⊕ il en aurait chargé l'information (propriété 1).

Donc tous les noeuds de x sont marqués d'un ⊕. L'information de chacun d'eux a donc été chargée (et recopiée à l'endroit voulu, comme nous allons le montrer maintenant).

La propriété 2 est donc vérifiée.

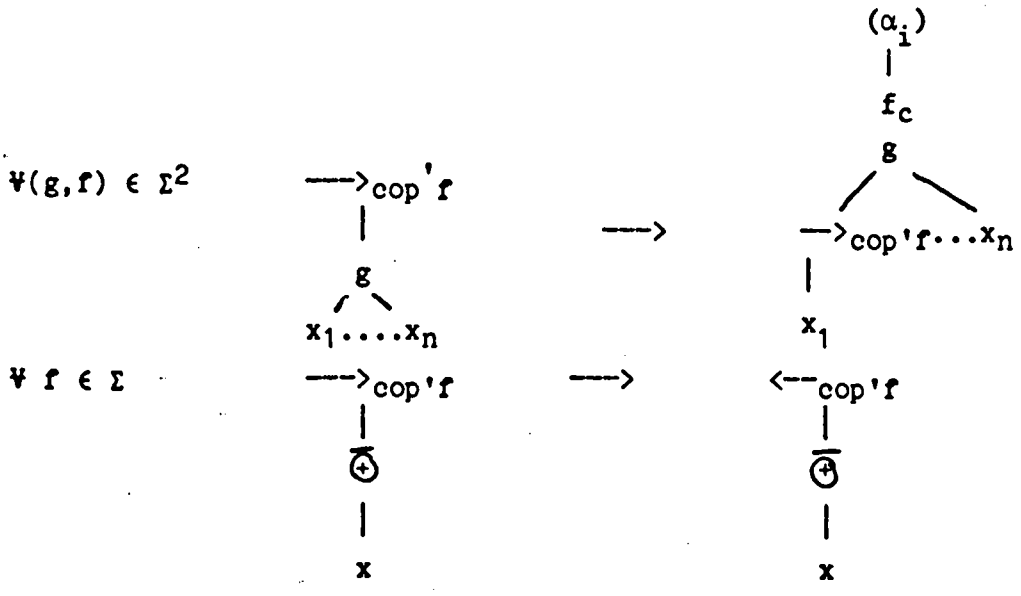
4. Remonter l'information au sommet

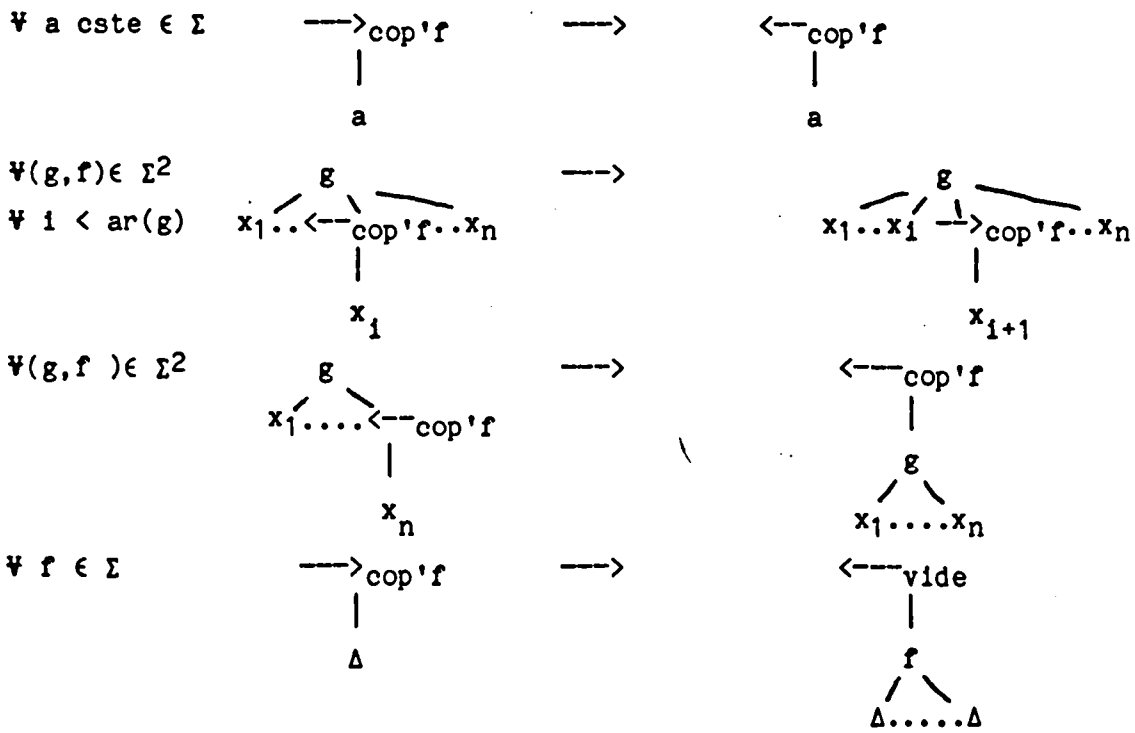
Il s'agit d'un système permettant à $\longrightarrow_{copf}$ de parcourir l'arbre. La dernière règle en sera :



5. Déposer l'information au noeud adéquat

Nous décrivons tout d'abord le système de réécriture effectuant ce travail, puis nous montrerons à l'aide de celui-ci que l'information doit être déposée sur le premier Δ non précédé d'une suite dc du sous-arbre de sommet Λ'





Il est clair que l'information est déposée sur le premier Δ relativement à l'ordre "sens de parcours".

Il faut noter que telle était bien sa destination :

- Si ce Δ a été engendré lors de la copie précédente :

C'est le fils gauche du symbole recopié lors de la phase précédente.

Or le symbole à déposer est le fils gauche du modèle de la phase précédente.

Les deux noeuds (modèle et Δ) sont donc bien correspondants.

- Si ce Δ provient d'une phase de copie antérieure :

La copie précédente a servi à compléter le sous-arbre remplaçant le immédiatement à gauche de celui dont nous nous occupons.

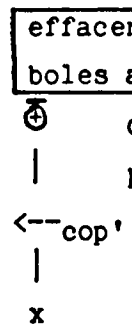
Si l'on suppose que cette copie précédente concernait le dernier noeu du sous-arbre du modèle correspondant, alors le symbole à déposer a bi comme destination le premier Δ de l'arbre (relativement à l'ordre "sens parcours").

On peut donc montrer par récurrence que si une copie a été faite bon endroit, il en est de même pour la suivante.

Ce qui démontre que toutes les copies se font où il faut, car la pr mière copie est "bien faite".

Pour terminer la description de la phase de copie d'un sous-arbre, nous devons définir le système de réécriture permettant, lorsque cette copie est terminée d'effacer les symboles auxiliaires $\oplus$ et $\overline{\oplus}$, désormais inutiles.

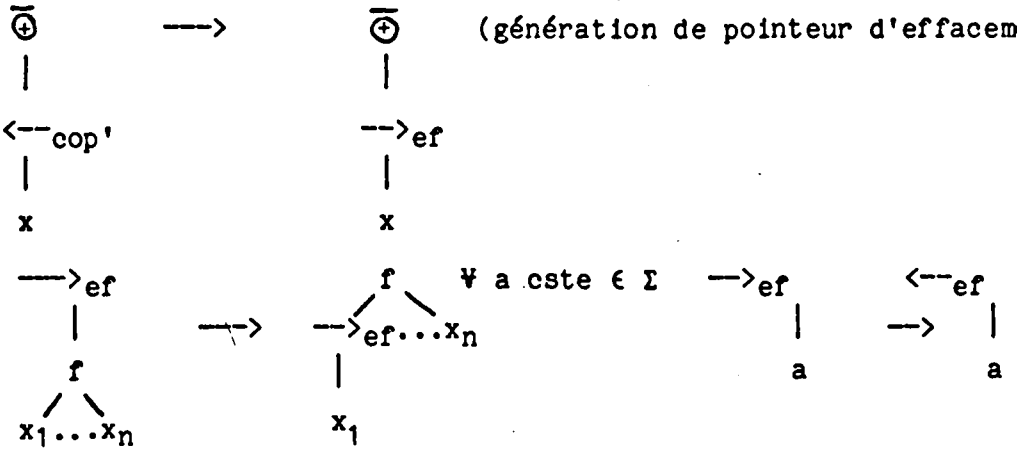
Cette primitive : effacer les symboles auxiliaires doit être déclenchée lorsqu'on obtient la configuration

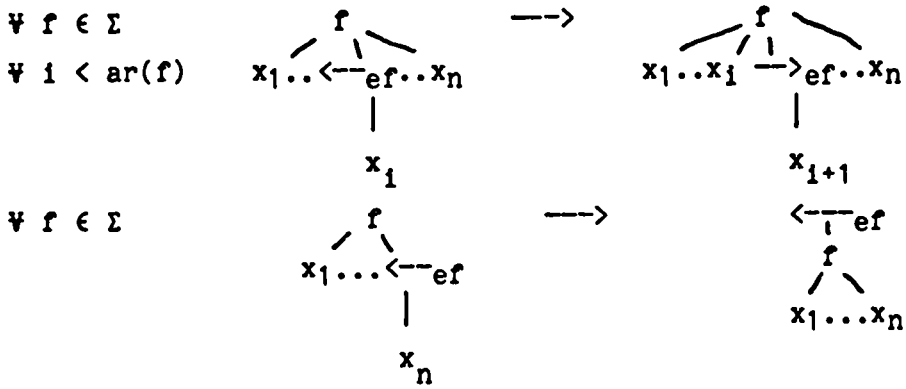


qui signifie que tous les noeuds de x ont été recopiés

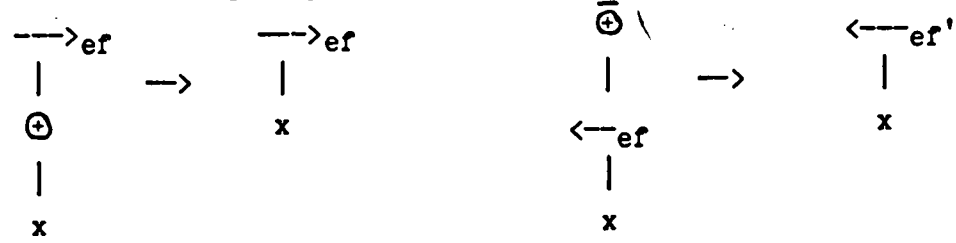
Nous effacerons tout d'abord les $\oplus$ et ensuite seulement le marqueur de sommet $\overline{\oplus}$ (ceci nous sera utile lors de l'application aux théories équationnelles).

Nous définissons par conséquent le système de réécriture :



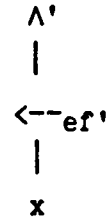


(4 familles de règles permettant de parcourir le sous-arbre)



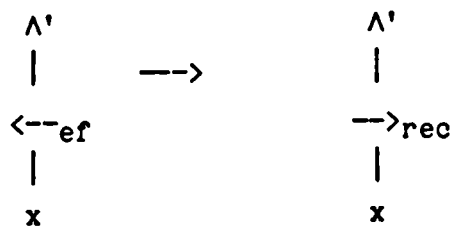
(règles d'effaçage)

Il faut de plus définir les règles permettant à $<\!\!---_{ef'}$ de regagner le sommet Λ'. Ce qui nous permet alors d'énoncer :

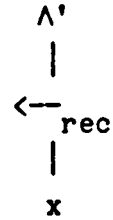
Proposition : La configuration  signifie que x ne contient plus ni ⊕ ni ⊖

dont la démonstration est claire, étant donnée la construction.

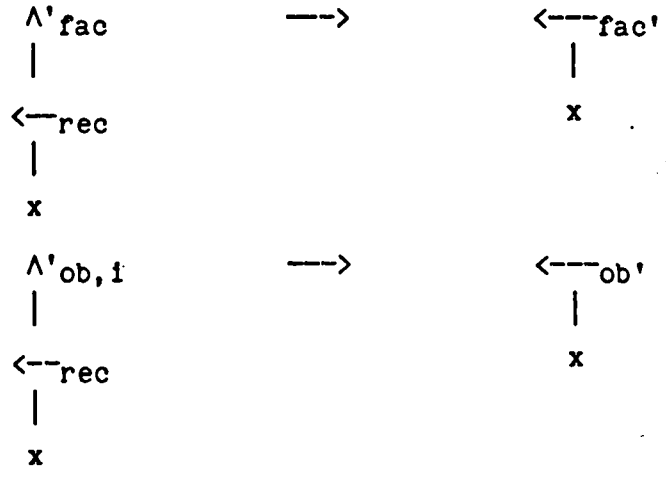
Nous pouvons alors relancer une phase de recherche, en créant la règle :



Une condition d'arrêt de la simulation est assurée par le lemme suivant :

lemme : La configuration  signifie que toutes les copies sont terminées.

dem : Le pointeur de recherche n'a donc rencontré aucun symbole dc. Puisque l'on est parti d'un linéarisé droit d'un arbre de $T(\Sigma)$, c'est donc que $x \in T(\Sigma)$ (d'après les définitions des systèmes de réécriture de ce chapitre), et que toutes les copies sont terminées.
Dès lors, on peut "dégeler" les pointeurs de dérivations, par exemple grâce aux règles suivantes :



(dans le cas où la structure de contrôle utilisée est celle "barrant" les symboles de Σ, une phase d'enlèvement des barres peut éventuellement précéder ce "dégel").

Nous avons donc pu définir un système de réécriture linéaire à gauche simulant une règle non linéaire à droite. L'utilisation de symboles auxiliaires nous a permis d'effectuer cette simulation de manière non ambigûe et sans interférence avec des applications d'autres règles.
(La non-ambiguîté de cette simulation est assurée par l'unicité du pointeur dans un sous-arbre isolé et par la nécessité de la présence du pointeur dans tous les membres gauches des règles générées).

Nous montrons maintenant plus formellement (algorithme) que cette simulation fonctionne correctement.

$\{\alpha_1\}$ et $\{\alpha_2\} = \{\alpha\}$
Linéarisation à droite
début
Engendrer un pointeur de recherche ; $\{\alpha_2\}$ et $\{\alpha_3\}$
Chercher un noeud à copier ; $\{\alpha_4\}$

Tant que (il reste des sous-arbres à copier) faire {i}

% recopier un sous-arbre %

Remonter le code au sommet Λ' ; {α_6}

Descendre jusqu'au noeud correspondant ; {α_7}

PREMIERE COPIE ; {α_8}

tant que (le sous-arbre n'est pas recopié) faire {j}

% Recopier un symbole %

Changer <---$_{vide}$ en --->$_{cop}$; {δ_1}

Chercher le symbole à recopier ; {δ_2}

Charger l'information ; {δ_3}

Remonter l'information au sommet ; {δ_4}

Descendre jusqu'au premier Δ ; {δ_5}

Déposer l'information ; {δ_6}

Remonter à vide ;

% symbole recopié %

fait {α_q}

Effacer les symboles auxiliaires ; {α_{10}}

Regénérer un pointeur de recherche ;

% Sous-arbre recopié % {α_5}

Chercher un noeud, où copier ;

fait ; {β_2} et {

Λ'

|

<—rec

|

u

}

Effacer les symboles auxiliaires et Regénérer un pointeur de contrôle

fin {β} = {β_1} et {β_2}

Avec :

{α_7}

Première copie

Début

Charger l'information et marquer le sommet du sous-arbre (⊖̄) ; {δ_1}

Remonter au sommet Λ' ; {δ_2}

Descendre jusqu'au premier Δ ; {δ_3}

Déposer l'information ; {δ_4}

Remonter à vide ;

fin {α_8}

L'algorithme de linéarisation à droite d'une règle est représenté par le programme de la page précédente.

Prouvons maintenant la correction de cet algorithme muni de ses assertions {α}-{on la configuration Λ , où Λ ∈ {Λ_{fac};Λ_{ob}}} et

{t ∈ $T(\Sigma \cup \{dc, fc, \alpha_i, \Delta\})^1_0$}

et est tel que :

- ces quatre symboles se présentent uniquement sous la configuration

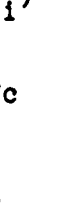
- n = * (dc,t)}

On considère que {α} = {α_1} et {α_2}

{β} = { on a la configuration <-- ' avec <-- ' ∈ { <-- $_{fac}$', <-- $_{ob}$'}} et

{u ∈ $T(\Sigma)^1_0$}

u est l'arbre t où l'on a recopié, à la place de chaque suite dc les sous-arbres de t dont le sommet correspondait à la suite (α_i) associée}



De même que précédemment, on considérera que {β} = {β_1} et {β_2}

Nous pouvons alors énoncer, vu les règles générant le pointeur de recherche $\longrightarrow_{rec}$, que :

{α} | engendrer un pointeur de recherche | {α_2} et {α_3}

avec $\{\alpha_3\}$ = {on a la configuration Λ' , $\Lambda' \in \{\Lambda'_{fac}, \Lambda'_{on}\}$}

Λ'
|
$\longrightarrow_{rec}$
|
t

est un résultat.

Montrons maintenant que :

$\{\alpha_2\}$ et $\{\alpha_3\}$ | chercher un noeud où copier | {on a un arbre de la forme Λ'avec $\Lambda' \in \{\Lambda'_{fac}, \Lambda'_{ob}\}$ et :

|
t

Si n = 0 t' = $\longleftarrow_{rec}$
|
t

Si n > 0 alors t' est l'arbre t où l'on a intercalé devant le premier dc de t (relativement à l'ordre sens de parcours) un pointeur $\longleftarrow_{rem}$ } = {α_4}.

est un résultat.

Démonstration :

a. n = 0

L'arbre t ne contient pas de symbole dc (donc $t \in T(\Sigma)^1_0$)

Or le système de réécriture associé à la primitive | chercher un noeud où copier | permet uniquement au pointeur $\longrightarrow_{rec}$ de parcourir t à la recherche d'un symbole dc.

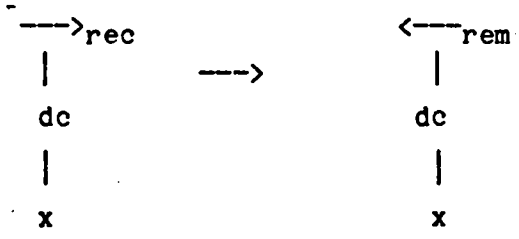
Comme t ne contient pas de dc, $\longrightarrow_{rec}$ parcourt intégralement t. Le rôle de la primitive s'arrête donc lorsque l'on a la configuration

Λ'
|
$\longleftarrow_{rec}$
|
t

(aucune règle de la primitive ne peut s'appliquer ici).

b. n ≠ 0

$\longrightarrow_{rec}$ parcourra t jusqu'à ce qu'il rencontre un symbole dc (donc le premier par rapport à l'ordre sens de parcours). La seule règle de la primitive applicable alors sera :



De plus, les pointeurs ne modifiant pas la structure ni les symboles de l'arbre qu'ils parcourent, et la primitive gérant uniquement un parcours de pointeur, il en résulte donc que l'on a, après application de cette règle (dernière règle de [chercher un noeud où copier]) la configuration Λ' (t), où t' n'est autre que t, avec insertion d'un pointeur $<---_{rem}$ devant le premier symbole dc de t.

Ce qui termine la preuve de cette primitive.

Nous arrivons alors au premier tant que :

$\{\alpha_4\}$ Tant que (il reste des sous-arbres à recopier) faire
{i}
Recopier un sous-arbre ;
$\{\alpha_5\}$
Chercher un noeud où copier ;
fait ;
$\{\beta_2\}$ et {on a la configuration Λ' (←rec, u)}

avec i = {α_4 où les t'(resp.t) sont remplacés par v'(resp. v)}
et {p = nombre des sous-arbres recopiés ; n = p + q ; q=≠ (dc,v)}

α_5 = {on a la configuration Λ (←rec, v) , v vérifiant α_2 ; p ← p + 1 ; q ← q - 1}

(en clair, la p^{ème} suite dc de t a été remplacée par le sous-arbre
| donc le code du sommet correspond à cette
(α_i) suite (α_i))
|
fc
|
Δ

Supposons pour l'instant que :

{i} | RECOPIER UN SOUS-ARBRE | {α_5} est un résultat

D'après la preuve précédente :

{α_5} | chercher un noeud où copier | {i} est un résultat

Démontrons alors le <u>tant que</u>

a. <u>Preuve d'arrêt</u>

Remarquons pour cela que { il reste des sous-arbres à recopier } <=> { on n'a pas la configuration Λ' } <=> {q = 0}
|
<--rec
|
x

Or chaque itération décremente strictement q.

Donc on n'effectuera qu'un nombre fini d'itérations.

b. <u>Preuve de correction</u>

Il nous faut vérifier la validité de l'invariant i

1. i est vrai à l'entrée du <u>tant que</u> :

Il suffit de poser v = t, v' = t'

donc q = n, p = 0

2. {i} et { Λ'} => {β_2} u { Λ' }
| |
<-rec <-rec
| |
v u

car : $q = 0$ donc $v \in T(\Sigma)^1_0$

$p = n$ donc tous les dc ont été remplacés par les sous-arbres correspondants (α_i)

fc

Δ

3. i est invariant, si l'on suppose, et nous le prouverons plus loin que

$\{i\}$ [Recopier un sous-arbre] $\{\alpha_5\}$ est un résultat

Montrons maintenant que :

{ on a la configuration } et $\{\beta_2\}$ [effacer les symboles auxiliaires et regenérer le pointeur de contrôle] $\{\beta\}$ est un résultat

Λ'

|

$<-_{rec}$

|

u

La seule règle de cette primitive efface la suite Λ' pour la remplacer

|

$<-_{rec}$

par $<--' \in \{ <--_{fac'} , <--_{ob'} \}$

Ces assertions sont donc valides.

Il nous reste donc à prouver que :

$\{i\}$ [RECOPIER UN SOUS-ARBRE] $\{\alpha_5\}$ est un résultat

Pour cela, montrons tout d'abord que :

$\{i\}$ [Remonter le code au sommet Λ'] $\{\alpha_6\}$ est un résultat

avec :

α_6 = on a la configuration Λ' (sommet) — $\xrightarrow{}_{ch}$ — dc — (α_i) — fc — v_1 ,

où v_1 est l'arbre v où on a enlevé la première suite dc de v. Cette suite se trouvant maintenant sous le (α_i) sommet Λ' — fc

Puisqu'on n'entre dans ce tant que que lorsque v contient au moins un symbole dc,

$\{i\}$ nous assure que le pointeur $\xleftarrow{}_{rem}$ est père du 1er dc de v.

Le système de réécriture correspondant à cette primitive peut donc entrer en application. Dès lors, le pointeur se chargera de remonter la suite jusqu'au sommet Λ'.

Ce qui valide les assertions.

Montrons que

$\{\alpha_6\}$ | descendre jusqu'au noeud correspondant | $\{\alpha_7\}$ est un résultat

avec :

α_7 = on a la configuration Λ' (sommet) — v_2 où v_2 est v_1 dans lequel on a intercalé devant le noeud de code (α_i), la suite $\xrightarrow{}_{ch}$ — dc — fc

le lemme de la page 22 entraîne le résultat.

Montrons que :

$\{\alpha_7\}$ | PREMIERE COPIE | $\{\alpha_8\}$ est un résultat

avec : α_8 = { On a la configuration Λ' (|, <--vide, |, v_3) , où v_3 est l'arbre v_1 dans lequel

1 : on a intercalé devant le noeud de code (α_i) un symbole $\bar{\Theta}$

2 : le premier Δ a été remplacé par f (Δ Δ) , si f est le symbole du noeud de code (α_i) }

Décomposons pour cela la preuve en plusieurs étapes.

et tout d'abord montrons que :

{α_7} [charger l'information et marquer le sommet du sous-arbre ($\bar{\Theta}$)] {γ_1} est un résultat

avec : γ_1 = { On a la configuration Λ' (|, v'_2) où v'_2 est l'arbre v_1 dans lequel

on a intercalé devant le noeud de code (α_i) la suite <---$_{copf,f}$ ($\bar{\Theta}$)

étant le symbole porté par ce noeud }

Cela résulte immédiatement de l'unique famille de règles de cette primitive

Montrons que :

{γ_1} [Remonter au sommet Λ'] {γ_2} est un résultat

avec : γ_2 = { On a la configuration Λ' (|, <--$_{copf}$, |, v''_2) où v''_2 est l'arbre v_1 dans lequel

on a intercalé devant le noeud de code (α_i) un symbole $\bar{\Theta}$, f étant le symbole porté par ce noeud }

Là encore, la définition de cette primitive, servant uniquement à remonter $--->_{copf}$ au sommet Λ' entraîne le résultat.

Montrons que :

$\{\gamma_2\}$ **descendre jusqu'au premier Δ** $\{\gamma_3\}$ est un résultat

avec :

γ_3 = On a la configuration Λ' où v'''_2 est l'arbre v''_2 dans lequel on a intercalé devant le premier symbole Δ le pointeur $\longrightarrow_{cop'f}$, où f est le symbole du noeud de code (α_i)

C'est ici le lemme de la page 22 qui nous assure du résultat.

Montrons que :

$\{\gamma_3\}$ **déposer l'information** $\{\gamma_4\}$ est un résultat

avec :

γ_4 : On a la configuration Λ' où $v^{(4)}_2$ est l'arbre v'''_2 dans lequel

1. Le pointeur $\longrightarrow_{cop'f}$ est remplacé par le pointeur $\longleftarrow_{vide}$
2. Le premier Δ de v'''_2 a été remplacé par $\overset{f}{\underset{\Delta\ \Delta}{}}$, f étant le symbole du noeud de code (α_i).

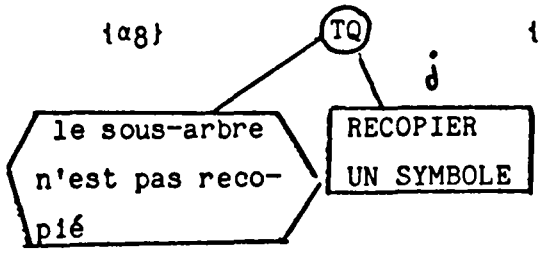
La seule famille de règles de cette primitive entraîne clairement le résultat.

De même, la définition de la primitive **remonter à vide** permet d'énoncer directement que

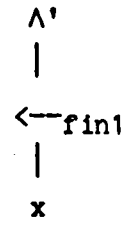
$\{\gamma_4\}$ **remonter à vide** $\{\alpha_8\}$ est un résultat

ce qui termine la preuve de **PREMIERE COPIE**

Il nous reste donc maintenant à prouver la boucle

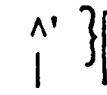


Rappelons que la condition d'arrêt de ce Tant Que peut s'exprimer, en termes de configuration d'arbres par :

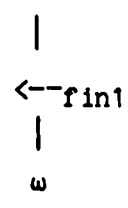
" on n'a pas la configuration 

"

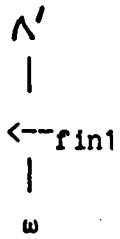
Supposons pour l'instant que :

{j et  } Recopier un symbole {j} soit un résultat

et posons :

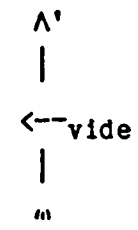
α_9 = { On a la configuration  où ω est l'arbre t dans lequel

les p premiers Δ de t ont été remplacés par les sous-arbres dont le sommet a pour code la suite (α_i) de chaque Δ.
Le sous-arbre dont le sommet correspond à la $p^{ième}$ +1 suite (α_i) de t est marqué au sommet par ⊖, tous ses descendants sont marqués d'un ⊕. }

{ Le noeud de code (α_i) a n_1 descendants. Les p_1 premiers descendants pour l'ordre sens de parcours ont été recopiés à l'endroit voulu.
Si $p_1 = n_1$, alors l'itération suivante conduira à la configuration 

}

Sinon on a la configuration



Dans les deux cas, ω est l'arbre v_3 où les p_1 premiers descendants du noeud de code (α_i) sont marqués d'un ⊕.

Ces p_1 noeuds ont été recopiés à l'endroit voulu.

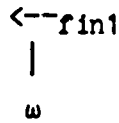
Supposons d'autre part pour l'instant que chaque itération, à l'exception de la dernière incremente p_1 d'une unité.

Preuve d'arrêt :

Puisque p_1 augmente de 1 à chaque itération et qu'il est borné supérieurement par n_1, on accomplira donc un nombre fini d'itérations $(n_1 + 1)$.

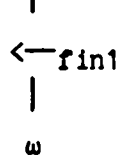
Preuve de correction :

On suppose pour l'instant que i j et ⌉ Λ' ⌈ Recopier un SYMBOLE ij est



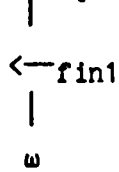
un résultat.

(α8 et ⌉ Λ') => j



En effet, α8 => p_1 = 0

{j et Λ'} => {α9}



Le Tant Que est donc prouvé, si l'on suppose que [RECOPIER UN SYMBOLE] l'est.

Montrons maintenant que :

$\{\alpha_9\}$ [effacer les symboles auxiliaires] $\{\alpha_{10}\}$ est un résultat, avec :

α_{10} = { On a la configuration Λ' (avec $\Lambda' \xrightarrow{ef} t_2$), où t_2 est l'arbre t dans lequel les p + 1 premiers Δ ont été remplacés par les sous-arbres correspondants aux codes associés. }

Cette primitive :

1. engendre un pointeur remplaçant $\xleftarrow{fin1}$ ($\xrightarrow{ef}$)
2. lui fait parcourir le sous-arbre de sommet Λ' de façon exhaustive, effaçant les symboles $\oplus$ et $\overline{\oplus}$ qu'il rencontre.

Ces considérations, ainsi que le lemme de la page 22, nous assure du résultat.

Nous avons ensuite clairement le résultat suivant :

$\{\alpha_{10}\}$ [Regénérer le pointeur de recherche] $\{i\}$

Il nous reste encore à prouver que

{ j et $\Lambda \xrightarrow{fin1} \omega$ } [Recopier un SYMBOLE] $\{j\}$ est un résultat

Procédons par étapes et montrons tout d'abord que

{ j et $\Lambda' \xrightarrow{fin1} \omega$ } [changer $\xleftarrow{vide}$ en $\xrightarrow{cop}$] $\{\delta_1\}$ est un résultat

avec :

$\delta_1 = \{ j$ dans l'expression duquel on remplace $<\!\!---_{vide}$ par $--\!\!>_{cop} \}$

Cette définition de δ_1 entraîne clairement le résultat.

Montrons que :

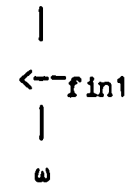
$\{\delta_1\}$ [chercher le symbole à recopier] $\{\delta_2\}$ est un résultat, avec :

$\delta_2 =$ Le noeud de code (α_i) a n_1 descendants. Les p_1 premiers par rapport à l'ordre sens de parcours ont été recopiés à l'endroit voulu et sont marqués d'un +.

Si $p_1 < n_1$, on a la configuration Λ' où ω' est l'arbre ω dans

lequel le $(p_1 + 1)^e$ descendant du noeud de code (α_i) est précédé d'un pointeur $--\!\!>_{cop'}$

Si $p_1 = n_1$ on a la configuration



La primitive :

1. recherche le symbole $\overline{\oplus}$, marquant le sommet du sous-arbre de code (α_i)
2. transforme le pointeur $--\!\!>_{cop}$ en $--\!\!>_{cop'}$ qui recherchera dans ce sous-arbre le premier noeud non marqué d'un $\oplus$.

   S'il en trouve un $(p_1 < n_1)$, le rôle de la primitive s'arrête.

   Sinon $(p_1 = n_1)$, $<\!\!---_{cop'}$ devient $<\!\!---_{fin1}$ qui remonte jusqu'au sommet Λ'.

(d'après les propriétés 1 et 2)

donc $\{\delta_1\}$ [chercher le symbole à recopier] $\{\delta_2\}$ est bien un résultat

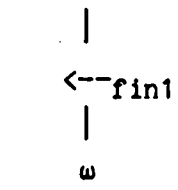
Montrons maintenant que :

$\{\delta_2\}$ [charger l'information] $\{\delta_3\}$ est un résultat, avec :

δ_3 =

Le noeud de code (α_i) a n_1 descendants. Les p_1 premiers pour l'ordre sens de parcours ont été recopiés à l'endroit voulu et sont marqués d'un +.

Si $p_1 = n_1$ on a la configuration



Si $p_1 < n_1$ alors le $(p_1 + 1)^e$ noeud est marqué d'un ⊕, lui-même précédé d'un pointeur $<---_{copf}$, si f est le symbole porté par le $(p_1 + 1)^e$ noeud

La seule famille de règles de cette primitive entraîne clairement le résultat.

Montrons que :

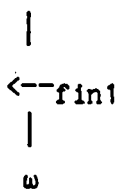
$\{\delta_3\}$ [Remonter l'information au sommet Λ'] $\{\delta_4\}$ est un résultat, avec :

δ_4 =

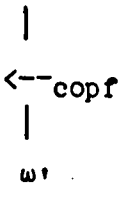
$p_1 \leftarrow p_1 + 1$

Le noeud de code (α_i) a n_1 descendants. Les p_1 premiers pour le sens de parcours ont été recopiés à l'endroit voulu et sont marqués d' un ⊕.

si $p_1 = n_1 + 1$ on a la configuration



Sinon on a la configuration



où ω' est l'arbre ω dans lequel le p^e descendant du noeud de code (α_i) est marqué d'un ⊕, et f est le symbole de ce noeud.

La démonstration est claire si l'on considère le travail effectué par cette primitive, consistant à remonter $--->_{copf}$ au sommet Λ'.

Montrons maintenant que :

$\{\delta_4\}$ [descendre jusqu'au premier symbole Δ] $\{\delta_5\}$ est un résultat, avec :

δ_5 = Le noeud de code (α_i) a n_1 descendants. Les p_1 premiers pour l'ordre sens de parcours ont été recopiés à l'endroit voulu et sont marqués d'un ⊕.

Sinon on a la configuration Λ' où ω'' est ω dans lequel un pointeur

ω''

$\longrightarrow_{cop'f}$ s'est intercalé devant le premier Δ.

Là encore, la structure du système de réécriture associé à cette primitive permet d'affirmer que les assertions sont valides.

Montrons que :

$\{\delta_5\}$ [déposer l'information] $\{\delta_6\}$ est un résultat, avec :

δ_6 = Le noeud de code (α_i) a n_1 descendants. Les p_1 premiers pour l'ordre sens de parcours ont été recopiés à l'endroit voulu et sont marqués d'un ⊕.

Si $p_1 = n_1$, on a la configuration Λ'

ω

$\longleftarrow_{fin1}$

Sinon les p_1 premiers descendants du noeud de code (α_i) sont marqués d'un ⊕ et sont recopiés à l'endroit voulu.

C'est ici le lemme b qui nous assure du résultat.

Il ne nous reste plus maintenant qu'à démontrer que :

$\{\delta_6\}$ [Remonter à vide] $\{j\}$ est un résultat

ce qui est clair, la primitive se contentant soit de remonter $\longleftarrow_{vide}$ au sommet Λ', soit si la copie est terminée de ne rien faire.

Cette démonstration termine donc la preuve de l'algorithme représentant la linéarisation à droite d'une règle de réécriture.

Conclusion

Il est somme toute simple de simuler linéairement une règle non linéaire à droite : il s'agit en fait de recopier certains sous-arbres à cetains endroits.

La plupart des règles définies dans ce chapitre servent en fait à contrôler cette copie, et à empêcher l'interférence d'autres règles.

CHAPITRE IV

Linéarisation à gauche d'une règle de réécriture

Linéarisation à gauche d'une règle

Introduction :

Nous avons vu que réduire la non-linéarité à droite d'une règle pouvait se résoudre en recopiant les sous-arbres ayant plusieurs occurences dans le membre droit de cette règle.

On était certain, dès qu'on lançait les opérations de copies, d'aboutir après un nombre fini de dérivations, au membre droit de la règle de départ.

Ce n'est plus le cas en ce qui concerne la linéarisation à gauche. Il faut en effet dans ce cas vérifier l'égalité de deux sous-arbres afin de savoir si la règle est applicable.

Selon la réponse à ce test, il faut :

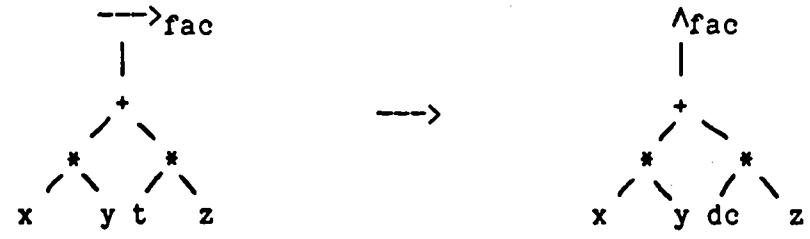
- Appliquer la règle et enchainer sur la phase de linéarisation à droite si les sous-arbres sont égaux.
- Revenir à l'arbre de départ et éviter de créer des cycles s'ils diffèrent.

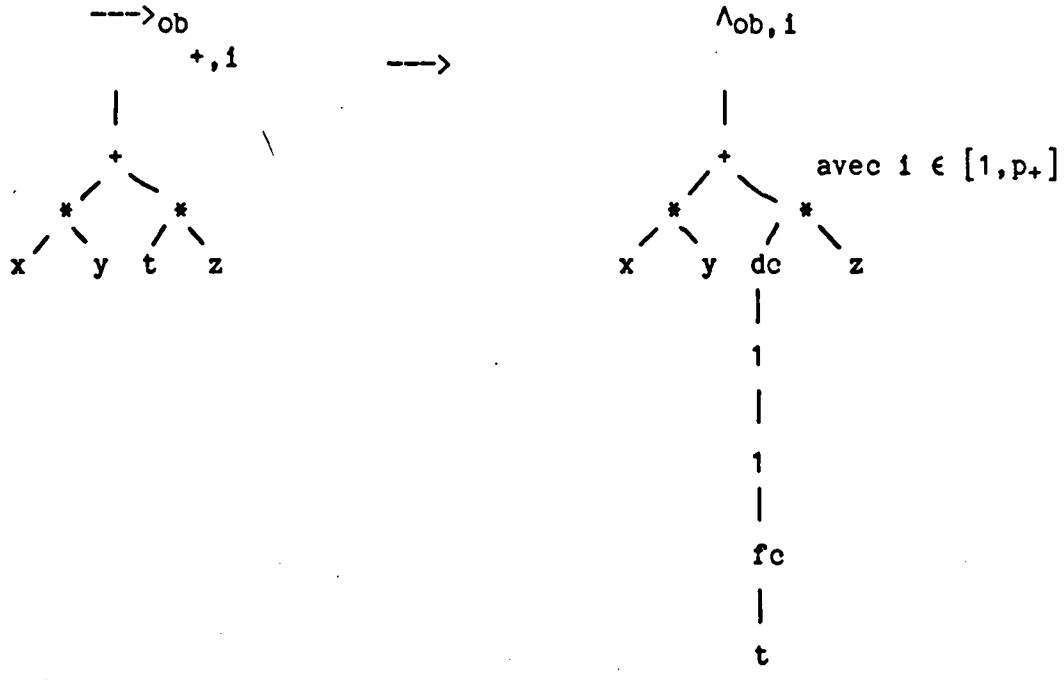
C'est dans ce second cas que se révèle le rôle essentiel des pointeurs de dérivation $\longrightarrow_{fac}$ et $\longrightarrow_{ob}$, et celui des structures de contrôle associées.

Mode opératoire :

Considérons l'exemple suivant :

Nous remplaçons cette règle non linéaire à gauche par les suivantes :





Ce premier pas nous permettra de lancer une suite de nouvelles règles de réécriture regroupées en primitives dont le but sera de comparer symbole par symbole les sous-arbres substitués en x et en t.

Si ceux-ci sont différents, (i.e : la règle remplacée n'est pas applicable), on regenerera les pointeurs de dérivation, de telle façon qu'on évite la création de cycles.
Sinon, on appliquera la règle de départ modifiée de telle façon qu'on puisse réduire une éventuelle non linéarité à droite par la méthode exposée précédemment.

Linéarisation à gauche

```
début
  Engendrer un pointeur de recherche ;
  Chercher un noeud à comparer ;
  Tant que ¬| fin 1 et ¬| diff faire
          % Comparer %
          Marquer le sommet du sous-arbre à comparer (⊗) ;
          Remonter le code au sommet Λ' ;
          Chercher le noeud correspondant ;
             % Première comparaison %
               Marquer le sommet ⊕ et charger l'information ;
               Remonter l'information au sommet ;
               Descendre jusqu'au sommet du sous-arbre à comparer ;
               Si ¬| diff alors Remonter à vide ;
                        sinon Remonter un indicateur de différence
               fin si
           % Fin de la première comparaison %
           COMPARER LES DEUX SOUS-ARBRES
           % Fin de comparer %
   fait
  Si ¬| diff alors appliquer la règle ;
  finsi
 Fin
```

Avec :

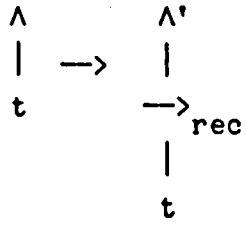
```
                COMPARER LES DEUX SOUS-ARBRES

   Début
    Si ¬| diff alors Aller chercher information suivante ;
    finsi
    Tant que ¬|FIN et ¬| diff faire
        Remonter l'information au sommet Λ' ;
```

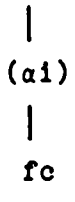
Redescendre jusqu'au sommet de l'arbre à comparer ;
Rechercher le noeud à comparer avec l'information ;
Si ¬| diff alors
Marquer ce noeud ;
Remonter à vide ;
Aller chercher information suivante ;
Sinon Remonter indicateur de différence ;
Finsi
Fait
Si ¬| diff alors
Effacement partiel ;
Regenerer un pointeur de recherche ;
Chercher un arbre à comparer ;
Sinon
Effacement total ;
Finsi ;
fin.

A la lecture de ces algorithmes on s'aperçoit qu'un certain nombre de primitives définies dans le chapitre précédent pourront être réutilisées plus ou moins directement. Elles ne seront donc pas toutes définies.

1. Générer un pointeur de recherche :
La seule règle est



2. Chercher un noeud à comparer :
Il s'agit, comme lors de la linéarisation à droite, de permettre à $--->_{rec}$ de repérer la première suite

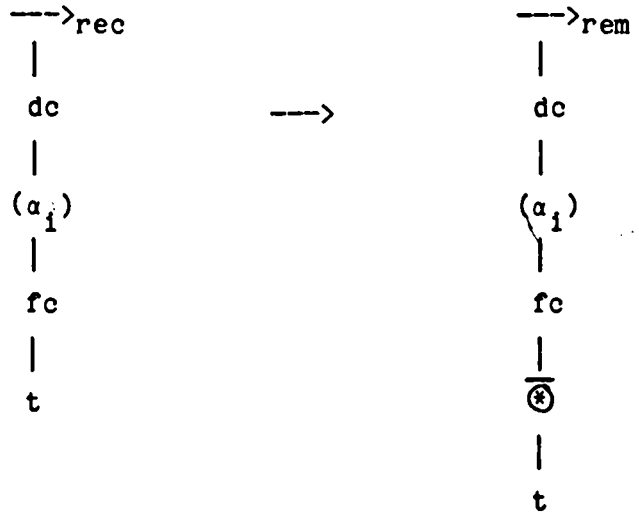


Le système correspondant est décrit page dans le chapitre III.

3. Marquer le sommet du sous-arbre à comparer :

On introduit les deux nouveaux symboles auxiliaires ⊛ et $\overline{⊛}$ servant à marquer les noeuds déjà comparés avec leur modèle (ou en cours de comparaison).

Cette action correspond à la famille de règles suivantes :



et servira à retrouver lors des étapes ultérieures de la simulation le sous-arbre à comparer.

4. Remonter le code au sommet :

Le système de réécriture est le même que celui décrit au chapitre III, nous ne le rappelons pas.

5. Chercher le noeud correspondant au code :

Il s'agit, comme dans l'algorithme de linéarisation à droite, de se servir de la suite (α_i) pour atteindre le sous-arbre modèle.

Le système de réécriture correspondant est décrit au chapitre III.

6. Première comparaison :

Similairement à [première copie] dans le chapitre précédent, nous avons séparé cette partie du reste de la comparaison (cela facilite entre autres l'écriture de l'algorithme).

Cette action se décompose en primitives conformément à l'algorithme suivant :

Première comparaison

début

Marquer le sommet du modèle ($\overline{⊕}$) et charger l'information ;

Remonter l'information au sommet ;

Descendre jusqu'au sommet de t ;

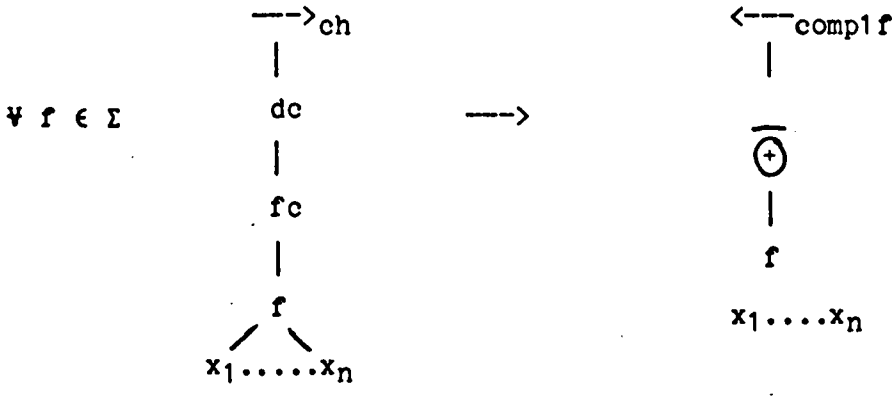
Si ¬ diff alors Remonter à vide ;
Sinon Remonter un indicateur de différence ;

Finsi

Fin.

6.1 Marquer le sommet et charger l'information :

Cette action est exprimée par la famille de règles :



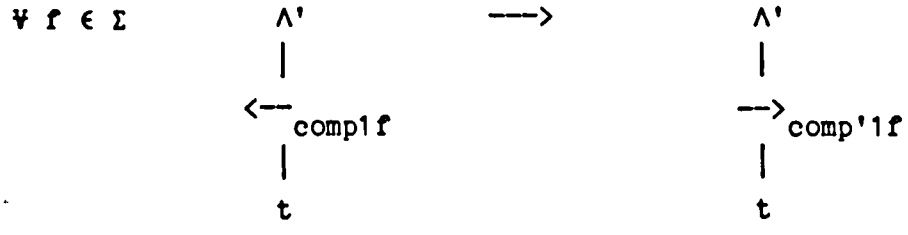
6.2 Remonter l'information au sommet :

Il s'agit uniquement de permettre à $\longrightarrow_{comp1f}$ de regagner le sommet Λ'. Ce genre de primitive a déjà été détaillé au chapitre III.

6.3 Redescendre l'information jusqu'au sommet de t :

Cette primitive peut être décomposée en trois parties :

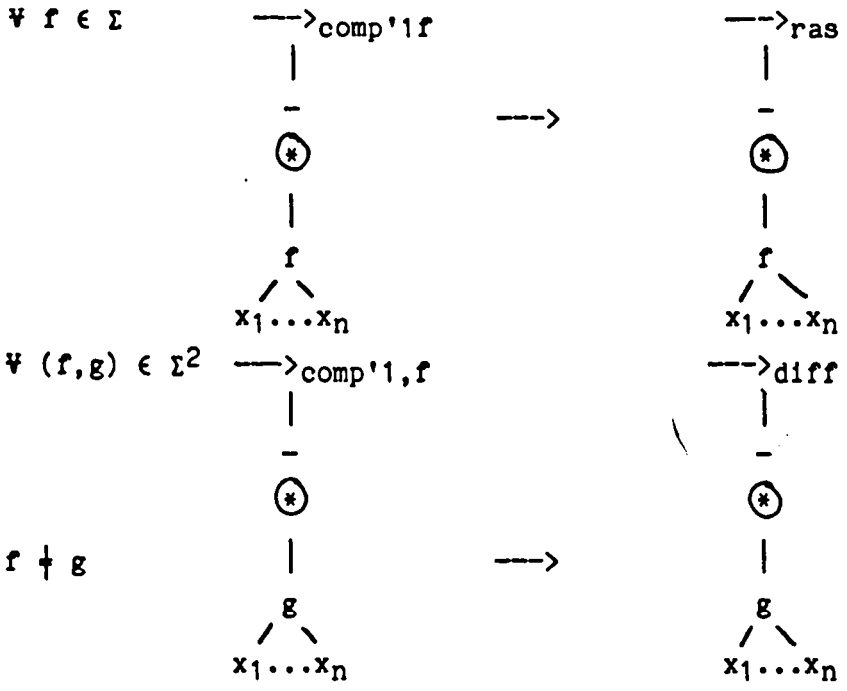
6.3.a : Engendrer un pointeur qui cherchera le sommet de t(marqué d'un $\overline{\otimes}$) :



6.3.b : Permettre à $\longrightarrow_{comp'1f}$ de parcourir l'arbre à la recherche du symbole $\overline{\otimes}$:

Il s'agit de règles habituelles de parcours d'arbre que nous ne détaillerons pas.

6.3.c : Rencontre du sommet de t :



C'est dans cette troisième phase qu'a lieu le test sur l'égalité des sommets de chacun des sous-arbres.

Cette partie de la primitive correspond au Si Alors Sinon de

Première comparaison

Il faut ajouter encore les règles permettant aux pointeurs $--->_{ras}$ et $--->_{diff}$ de rejoindre le sommet Λ'. (Nous ne les décrivons pas).

Ainsi, à l'aide des primitives définies jusqu'à présent :

- Nous savons comparer les sommets des deux sous-arbres
- Nous possédons un test (la présence sous Λ' d'un pointeur $<---_{ras}$ $<---_{diff}$) qui nous indique s'il faut continuer la comparaison ou si l'on peut l'arrêter.

Dans ce second cas (les deux sous-arbres diffèrent) il nous faut revenir à l'arbre de départ, c'est-à-dire effacer tous les symboles de l'ensemble {⊕, $\overline{⊕}$, ⊛, $\overline{⊛}$, dc, (α_i), fc} présents sous Λ', puis effacer Λ' en régénérant le pointeur de dérivation correspondant.

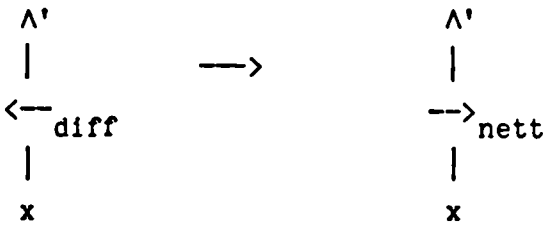
Remarque :

A ce stade des définitions, il n'y a pas, sous Λ', de symbole ⊕ ou $\overline{⊕}$. Ce sera le cas uniquement lors des comparaisons ultérieures. Nous décrivons donc ici le système général d'effacement applicable à tout moment lorsqu'une différence sera constatée.

Nous procéderons en quatre étapes :

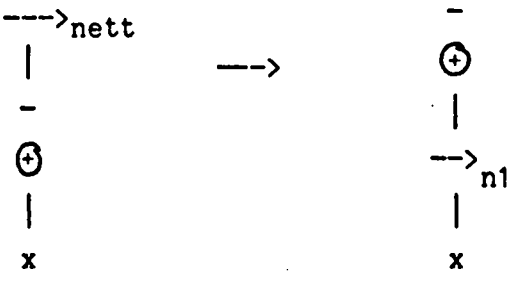
1. Effacer ⊕ et $\overline{⊕}$
2. Effacer ⊛ et $\overline{⊛}$
3. Effacer dc, (α_i), fc
4. Effacer Λ' et restituer le sous-arbre à son contexte.

Engendrons tout d'abord un pointeur de "nettoyage", grâce à la règle suivante :



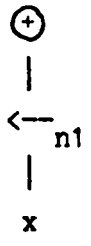
auquel nous permettons de parcourir l'arbre à la recherche de $\overline{⊕}$ (nous ne décrivons pas ces règles).

Etape 1 : (Effacer les ⊕, puis $\overline{⊕}$) :

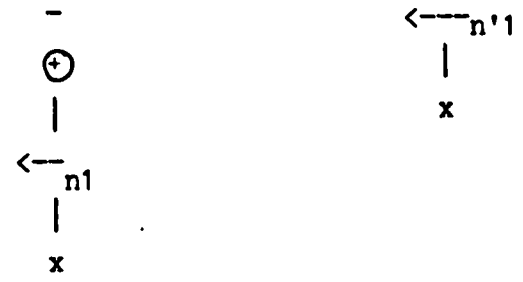


Similairement à ce que nous avons fait au chapitre précédent, , nous forçons $--->_{n1}$ à effacer les ⊕

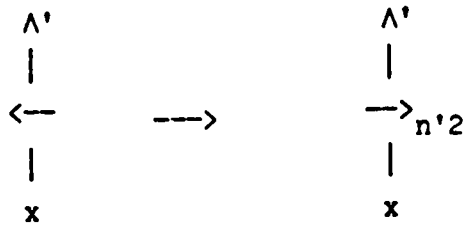
Nous obtenons alors à la fin de cette phase la configuration



On efface alors $\overline{⊕}$, engendrant par la même occasion un nouveau pointeur $<---_{n'1}$ dont l'unique rôle sera de rejoindre Λ', afin de signaler la fin de cette pemière étape :

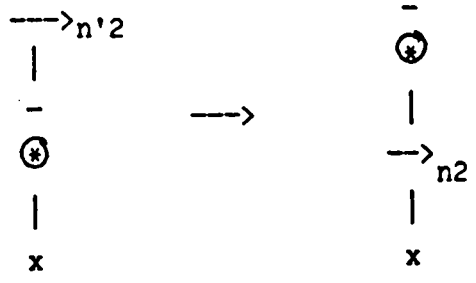


Etape 2 : Après avoir remonté <---$_{n'1}$ sous le sommet, nous changeons de pointeur :



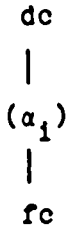
dont le rôle sera de rechercher le sous-arbre dont le sommet est marqué d'un $\overline{\circledast}$.

On crée la règle

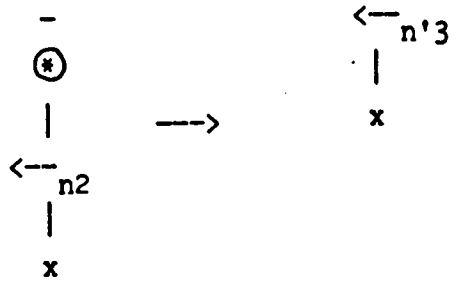


--->$_{n2}$ est muni de règles le forçant à effacer les $\circledast$.

Lorsque cette phase est terminée, il nous reste encore à effacer les suites

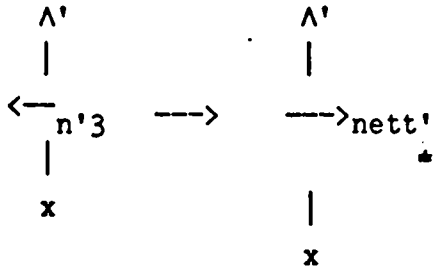


On crée donc la règle :



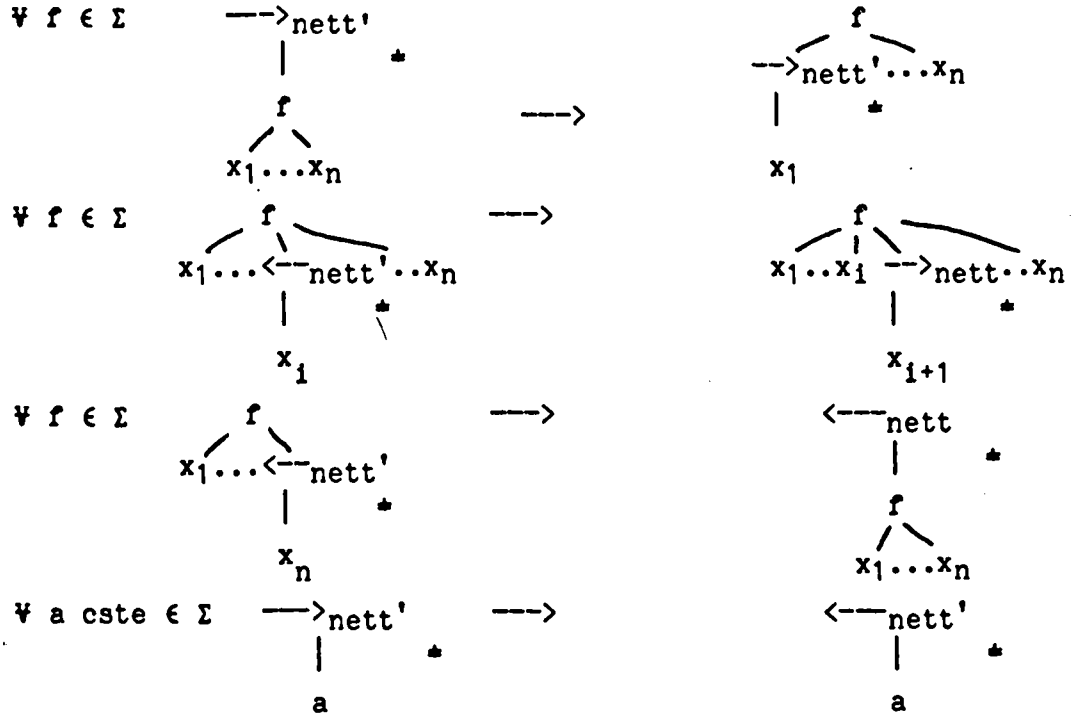
<---$_{n'3}$ est muni de règles lui permettant de rejoindre le sommet Λ'.

On crée alors la règle :



$--->_{n3}$ est muni de règles qui lui permettent de parcourir le sous-arbre de Λ' en effaçant tous les symboles n'appartenant pas à Σ. (Remarquons qu'il ne reste alors sous Λ' que les suites $\begin{matrix} dc \\ | \\ (\alpha_i) \\ | \\ fc \end{matrix}$ qui ne sont pas dans Σ)

On peut alors décrire le système de réécriture de la manière suivante (en effaçant les suites entières d'une seule règle).



(règles usuelles de parcours).

$\forall$ n $\leq$ { hauteur du membre gauche de la règle de départ } $\longrightarrow_{nett'}$ dc $(\alpha_i)_{1 \leq i \leq n}$ fc x $\longrightarrow$ $\longrightarrow_{nett'}$ x

$\forall$ $\alpha_i \leq$ { arité maximale des éléments de Σ }

(puisque n et les α_i sont bornés, cette dernière famille de règles est finie).

Ainsi, lorsqu'on obtient la configuration Λ' $\leftarrow_{nett'}$ t on peut assurer que $t \in T(\Sigma)$

si l'on est parti d'un terme de $T(\Sigma)$

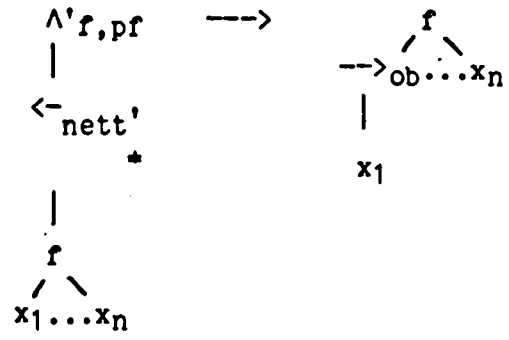
Revenons maintenant à l'effacement de Λ', qui doit intervenir lorsqu'on a obtenu la configuration Λ' $\leftarrow_{nett'}$ x

Rappelons que $\Lambda' \in \{\Lambda'_{fac}\} \cup \{\Lambda'_{f,i} / f \in \Sigma, i \in [1, p_f]\}$

Ce qui nous entraîne à créer les différentes règles suivantes :

$\forall$ f $\in \Sigma$: Λ'_{fac} $\leftarrow_{nett'}$ f $(x_1 \ldots x_n)$ $\longrightarrow$ f $\rightarrow_{fac \ldots x_n}$ x_1 — la dérivation facultative, on fait descendre le pointeur, afin de ne pas essayer d'appliquer la même règle au même endroit.

$\forall$ f $\in \Sigma$ $\forall$ i < p_f : $\Lambda'_{f,i}$ $\leftarrow_{nett'}$ f $(x_1 \ldots x_n)$ $\longrightarrow$ $\Lambda_{f,i+1}$ f $(x_1 \ldots x_n)$ — La $i^{è}$ règle dont le membre gauche commence par f n'est pas applicable, on va essayer la $(i+1)^{e}$

$\forall f \in \Sigma$ _ 

Aucune règle dont le membre gauche commence par f n'est pas appliquable, on fait descendre le pointeur

ce qui termine la description à accomplir lorsque les deux arbres à comparer diffèrent.

Revenons au cas où ces deux sommets sont égaux.

Nous avions alors engendré un pointeur $--->_{ras}$, et défini les règles de réécriture lui permettant de remonter sous le sommet Λ'.

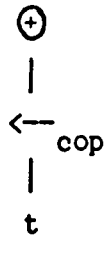
Nous devons donc maintenant décrire l'action [COMPARER 2 SOUS-ARBRES], et tout d'abord définir la primitive [aller chercher information suivante], que l'on peut diviser en trois parties :

1. Engendrer un pointeur recherchant le sommet du modèle (⊕̄)
2. Rechercher le premier noeud non marqué d'un ⊕
3. Charger l'information correspondant à ce noeud.

On voit donc que l'on peut utiliser sans modifications le système de réécriture défini pour la copie, dans le cas de la linéarisation à doite (voir chapitre précédent).

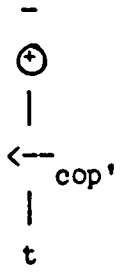
Nous reprenons donc ici cette primitive, sans avoir besoin de la détailler.

Ce qui était précédemment la condition d'arrêt de la copie, à savoir la configuration -



devient donc la condition d'arrêt de la comparaison, et on a le lemme suivant :

lemme : la configuration



nous permet d'affirmer que les deux sous-arbres à comparer sont égaux.

Démonstration :

Cette configuration nous apprend que tous les noeuds de t ont été comparés avec succès aux noeuds correspondants du second sous-arbre. Comme de plus t et ce second sous-arbre sont des termes clos, les deux sous-arbrres sont donc égaux.

La primitive suivante ; | Remonter l'information au sommet | correspond à celle que nous avons défini au chapitre III, il n'est donc pas nécessaire de la réécrire ici.

Voyons maintenant | Redescendre jusqu'au sommet de t |

Rappelons les conditions initiales :

- sous le sommet, nous avons un pointeur $<--_{cop}$, g étant le dernier symbole chargé
- le sommet de t est marqué d'un $\overline{\circledast}$, tous les noeuds de t déjà comparés sont marqués d'un $\circledast$. (Ces noeuds sont les premiers relativement à l'ordre sens de parcours).

Lors de la linéarisation à droite, il nous suffisait d'effectuer un parcours de l'arbre à la recherche du premier Δ, puisque nous étions assuré que c'est là qu'il fallait déposer l'information.

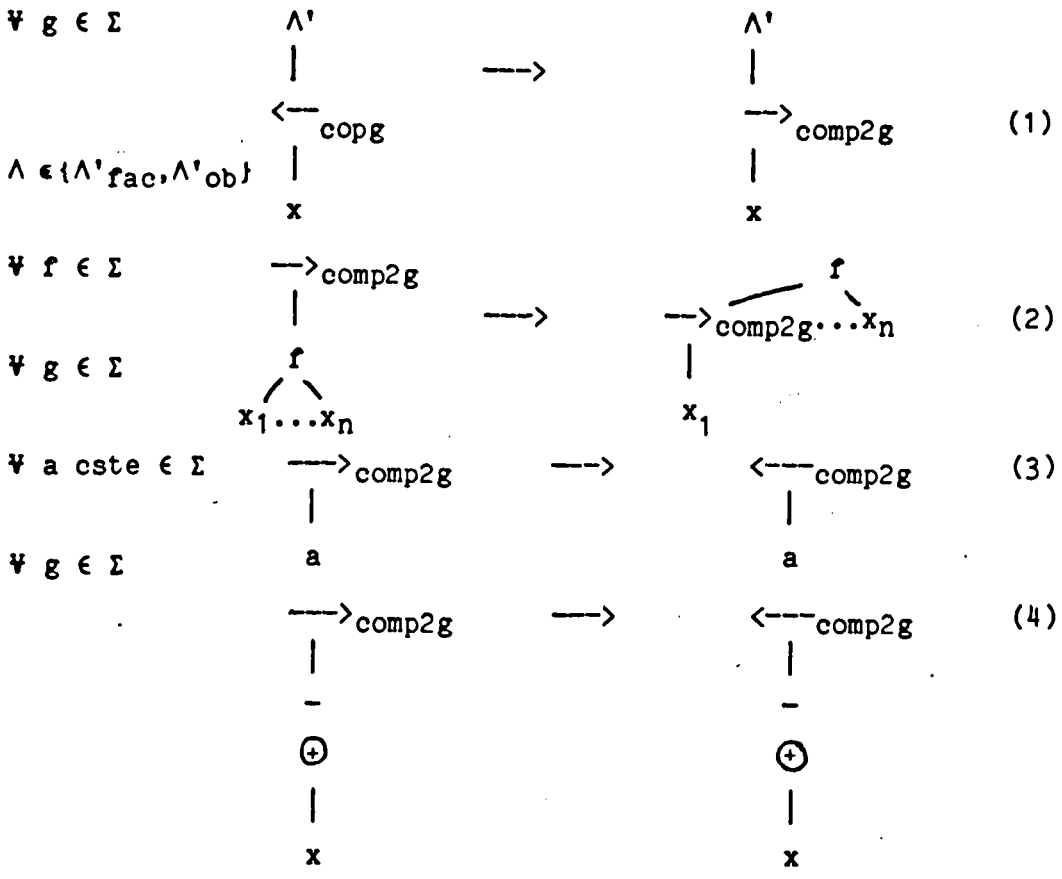
Ce n'est plus le cas maintenant.

Il nous faudra :

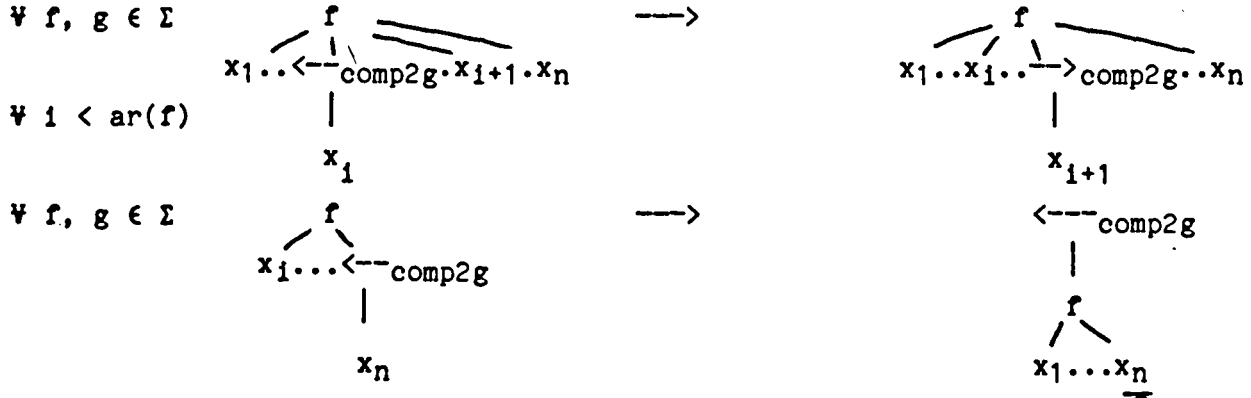
1. générer un pointeur a) conservant l'information de $<--_{copg}$
   b) recherchant le $\overline{\circledast}$ (sommet de t)
2. transformer ce pointeur en un autre qui cherchera dans t le premier noeud non marqué d'un $\circledast$

3. comparer g au noeud repéré.

Ce que nous traduisons en termes de systèmes de réécriture :



(il n'est pas nécessaire d'explorer le modèle)



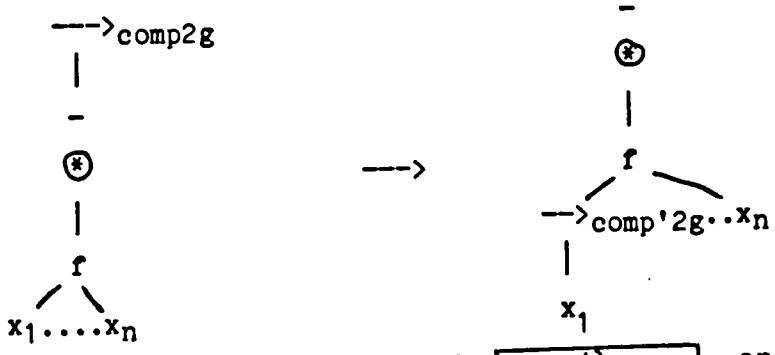
Ces règles définissent le parcours de $\longrightarrow_{comp2g}$, à la recherche de $\overline{\circledast}$.

En particulier, la famille (4) exprime le fait que les deux arbres à comparer sont distincts* (en effet, ils correspondent à deux instanciations de variables dans la règle non linéaire à gauche d'où nous sommes partis).

* ($\neg$ ($x \subset t$) et $\neg$ ($t \subset x$))

Examinons maintenant la rencontre de $\longrightarrow_{comp2g}$ et $\overline{\circledast}$.

$\forall$ f, g $\in \Sigma$

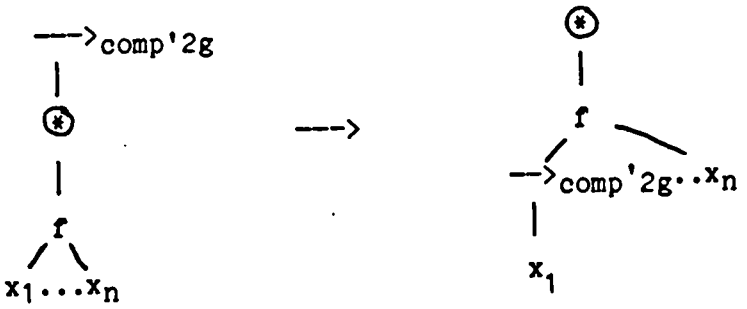


(f ayant été comparé au sommet de x lors de **première comparaison**, on peut passer directement au noeud suivant)

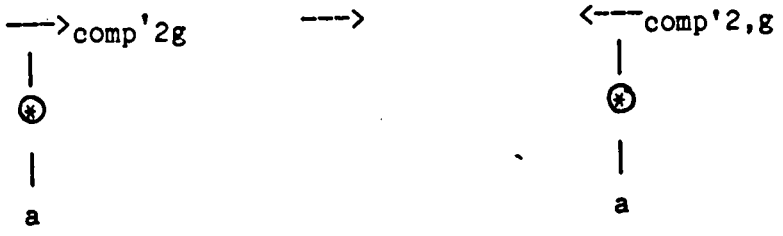
Définissons les règles permettant de repérer le premier noeud non marqué d'un $\circledast$.

(Elles sont similaires à celles décrites au chapitre précédent, qui permettaient de rechercher le premier noeud non marqué d'un $\oplus$).

$\forall$ f, g $\in \Sigma$

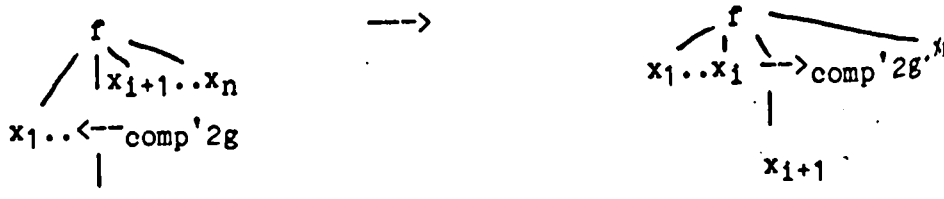


$\forall$ a cste $\in \Sigma$



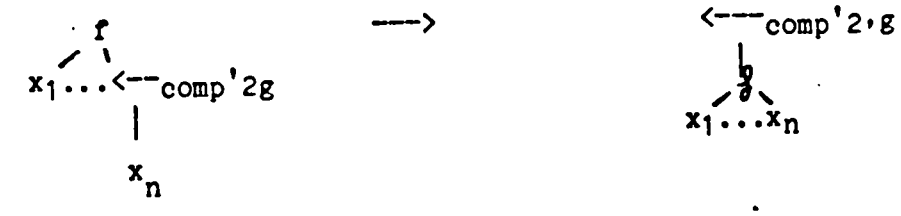
$\forall$ g $\in \Sigma$

$\forall$f, g $\in \Sigma$



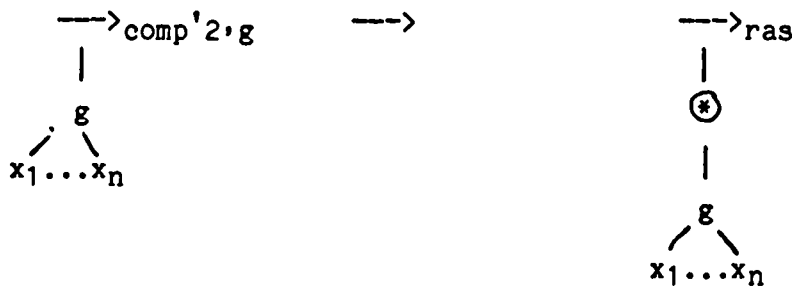
$\forall$ i <ar(f)

$\forall$ g $\in \Sigma$

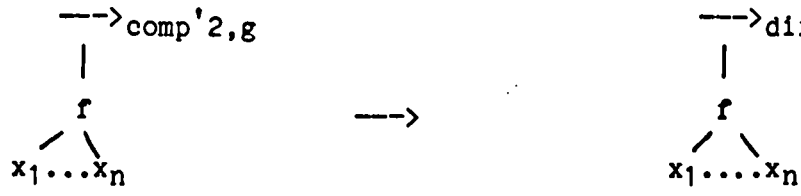


$\forall$f $\in \Sigma$ u{⊛}

$\forall\ g \in \Sigma$



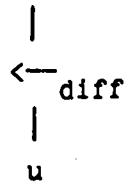
$\forall\ f \neq g \in \Sigma$



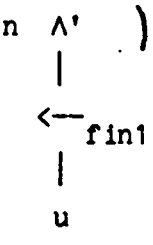
Pour terminer la description de [COMPARER DEUX SOUS-ARBRES], il nous reste à définir la primitive [effacement des symboles auxiliaires]

Deux cas peuvent se présenter :

1. La comparaison a échoué (i.e : on a obtenu la configuration



2. La comparaison est réussie (i.e : on a obtenu la configuration



Dans le premier cas, la règle n'est pas applicable. Il faut "dégeler" le pointeur de dérivation et revenir à l'arbre auquel on est parti (à la position du pointeur de dérivation près). Il faut donc effacer $\oplus$, $\overline{\oplus}$, $\circledast$, $\overline{\circledast}$, dc, fc, (α_i).

Dans le second cas, on n'effacera que les symboles $\oplus$, $\bar{\oplus}$, $\circledast$, $\bar{\circledast}$.

Nous définissons donc deux primitives d'effacement : effacement total

et effacement partiel.

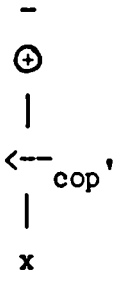
Effacement total a été décrit lors de la définition de première comparaison

Décrivons effacement partiel :

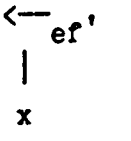
Cette primitive se décompose en plusieurs phases :

- Effacer les $\oplus$, puis le $\bar{\oplus}$, les $\circledast$ et enfin $\bar{\circledast}$.

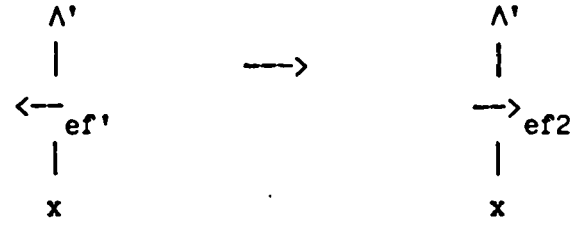
On entrera dans cette primitive lorsqu'on aura obtenu la configuration



On utilisera alors, dans les phases d'effacement des $\oplus$ et du $\bar{\oplus}$, le système décrit précédemment, à l'issue duquel on obtient la configuration

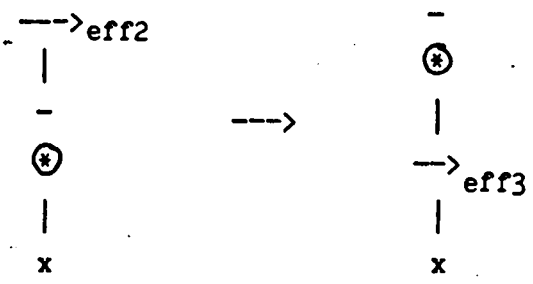


en créant les règles permettant à $<\!\!-\!\!-_{ef'}$ de regagner le sommet $\wedge'$, nous pouvons ensuite engendrer un nouveau pointeur $-\!\!>_{ef2}$, par la règle :



Ce nouveau pointeur est muni alors de règles lui permettant de parcourir l'arbre à la recherche du sous-arbre dont le sommet est marqué d'un $\bar{\circledast}$.

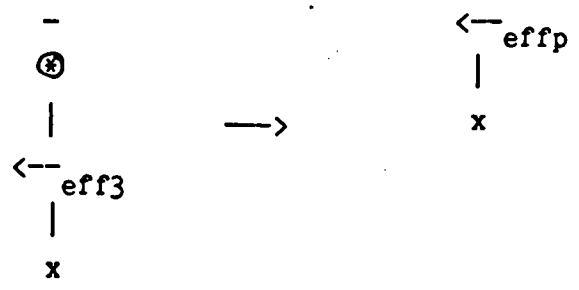
On crée donc, outre ces règles de parcours, la règle suivante :



A l'aide de règles en tout point similaires à celle effaçant les ⊕ et le $\overline{⊕}$, on permet alors à $—>_{eff3}$ d'effacer les ⊛.

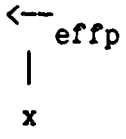
Pour terminer on crée :

a. La règle effaçant le $\overline{⊛}$:

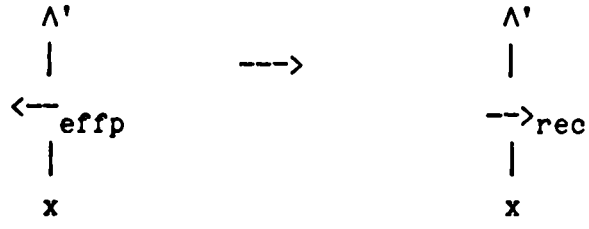


b. Les règles permettant à $<—_{effp}$ de regagner le sommet Λ'.

Ainsi, la configuration Λ' nous assure que tous les ⊕ , $\overline{⊕}$, ⊛ , $\overline{⊛}$, ont été effacés dans le sous-arbre.



On peut alors regénérer un pointeur de recherche afin de résoudre d'éventuels autres cas de non-linéarité :



Il nous reste encore à définir la primitive | application de la règle |

Il faudra l'exécuter quand tous les tets de comparaison auront été positifs. (i.e : quand la règle de départ était applicable).

Le sous-arbre de Λ' pourra alors être remplacé par le membre droit de cette règle de départ.

Or celui-ci peut n'être pas linéaire non plus. Il serait donc judicieu
d'embrayer alors sur l'algorithme de linéarisation à droite qui fait l'ob
jet du chapitre précédent.
Nous repoussons donc la description de cette primitive au chapitre suivant
et admettrons sa correction et prouvons l'arbre programmatique correspon
dant à l'algorithme de linéarisation à gauche.

```
{α}
  Linéarisation à gauche
   Début
     Engendrer un pointeur de recherche ;  {α1}
     Chercher un noeud à comparer ;        {α2}
     Tant que ¬| fin 1  et ¬| diff  faire {1}
              % Comparer %
              Marquer le sommet du sous-arbre à comparer ( ⊛̄ ) ; {α4}
              Remonter le code au sommet Λ' ; {α5}
              Chercher le noeud correspondant ; {α6}
                  % Première comparaison %
                    Marquer le sommet ( ⊛̄ ) et charger l'information ;{α7}
                  Remonter l'information au sommet ;  {α8}
                   Descendre jusqu'au sommet du sous-arbre ; {α9}
                  Si ¬| diff alors Remonter à vide ;
                                  Sinon Remonter un indicateur de différence ;
                   Finsi {α10}
             % Fin de la première comparaison %
            COMPARER LES DEUX SOUS-ARBRES
              % Fin de Comparer %
       Fait  {α3}
     Si ¬| diff alors  appliquer la règle ;
     Finsi
  Fin          {β}
Avec :
{α10} COMPARER LES DEUX SOUS-ARBRES
        Début
          Si  ¬| diff  alors Aller chercher information suivante ;
        Finsi  {α11}
```

```
tant que ¬FIN et ¬diff faire {i1}
    Remonter l'information au sommet Λ' ; {α13}
    Redescendre jusqu'au sommet de l'arbre à comparer ; {α14}
    Rechercher le noeud à comparer avec l'information ; {α15}
Si ¬diff alors
        Marquer ce noeud ; {α10}
        Remonter à vide ; {α17}
        Aller chercher information suivante ;
        Sinon Remonter indicateur de différence ;
Finsi
Fait {α12}
Si ¬diff alors
        Effacement partiel ;
        Regenerer un pointeur de recherche ;
        Chercher un arbre à comparer ;
        Sinon
         Effacement total ;
    finsi ;
fin
```

Posons :

α = { On a un arbre de la forme $\Lambda \backslash t$, où $t \in T(\Sigma \cup \{dc, (\alpha_1), fc\})^1_0$ et est tel qu'il existe $t_1 \in T(\Sigma)^1_0$ dont t soit un linéarisé gauche indicé.
Il existe une règle g -> d du système R de départ telle que t_1 soit unifiable avec g à l'occurence ε.
n = * (dc,t) }

β = { Si tous les sous-arbres à comparer dans t sont égaux, on a la configuration

Λ
|
$\longrightarrow_{rec'}$
|
t_2

où $t_2 \in T(\Sigma \cup \{dc, (\alpha_i), dc\})^1_0$ et il existe $t_3 \in T(\Sigma)^1_0$ tel que t_2 soit un linéarisé droit de t_3, et $t_1 \xrightarrow[(g \to d)/\epsilon]{R} t_3$

Sinon
Si $\Lambda = \Lambda_{fac}$, on obtient l'arbre $t' = t \cap T(\Sigma)^1_0$ où l'on a intercalé entre le sommet de t et son premier fils le pointeur de contrôle $\longrightarrow_{fac}$.
Si $\Lambda = \Lambda_{ob,i}$ $i < p(t/\epsilon)$ on obtient la configuration Λ | $_{ob,i+1}$ t 1

Si $\Lambda = \Lambda_{ob,p(t/\epsilon)}$ on obtient l'arbre t_1 où l'on a intercalé entre le sommet et son premier fils un pointeur $\longrightarrow_{ob}$ }

soit α_1 = { On a la configuration Λ' / $\longrightarrow_{rec}$ / t , t' vérifiant les conditions exprimées dans α . }

De façon évidente, $\{\alpha\}$ [engendrer un pointeur de recherche] $\{\alpha_1\}$ est un résultat.

Soit α_2 = { Si n = 0, on a la configuration Λ' / $\longleftarrow_{rec}$ / t , t vérifiant α

Sinon on a la configuration Λ' , où t' est t dans lequel un pointeur $\longrightarrow_{rec}$ s'est intercalé devant le premier symbole "dc" de t(relativement à l'ordre sens de parcours)

Montrons que $\{\alpha_1\}$ [chercher un noeud à comparer] $\{\alpha_2\}$ est un résultat

a. Si n = 0, $\longrightarrow_{rec}$ a parcouru t sans rencontrer de "dc". Il retourne donc sous Λ' sans avoir modifié t.

b. Sinon, $\longrightarrow_{rec}$ s'arrête sur le premier symbole "dc" qu'il rencontre d'où le résultat

Passons maintenant à la preuve du premier tant que

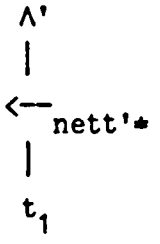
Procédons pour cela en deux étapes.

Dans un premier temps, nous supposerons que :

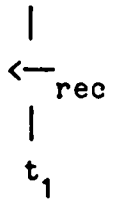
$\{i\}$ et ($\neg fin_1$ et $\neg diff$) [COMPARER] $\{i\}$ est un résultat, en posant :

p est le nombre de sous-arbres déjà comparés avec leur modèle.

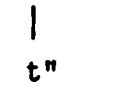
Si le sous-arbre fils du $p^{ème}$ "fc" (relativement au sens de parcours) est différent de son modèle, on a la configuration



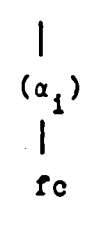
Sinon, si p = n on a la configuration



si p < n, on a la configuration



où t" est l'arbre t dont les p premières séquences

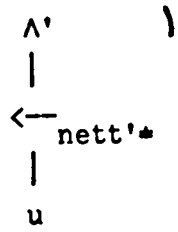


ont été effacées et où un pointeur $\longrightarrow_{rec}$ a été intercalé devant le $(p+1)^e$ "dc" de t (qui est le premier "dc" de t".

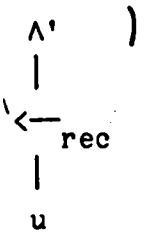
et prouvons le tant que :

Rappelons que les deux conditions formant le prédicat s'expriment sous forme de configuration d'arbres de la manière suivante :

diff <=> (on a la configuration



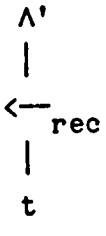
fin_1 <=> (on a la configuration



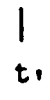
a. Montrons que α_2 => i

Pour cela il suffit de donner à p la valeur zéro.

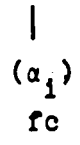
Si n = 0, on a bien la configuration



Sinon, on a la configuration



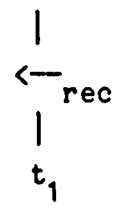
. t' est bien l'arbre t où aucune séquence



n'a été effacée et le pointeur $\longrightarrow_{rec}$ s'est intercalé devant le $(p + 1)^e$ symbole dc de t.

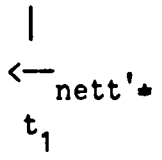
b. Posons

Si tous les sous-arbres de t à comparer sont égaux à leur modèle, alors on a la configuration



t_1 défini dans {α}

Sinon on a un arbre de la forme

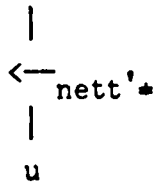


et montrons que

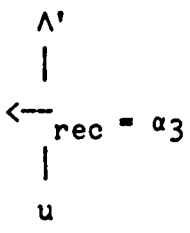
i et ¬(¬diff et ¬fin_1) => α_3

Ce qu'on écrit plus clairement :

i et (on a soit la configuration



, soit la configuration



) $\Rightarrow$ α_3

On a bien alors le résultat voulu, car u = t_1 d'après i et les configurations considérées.

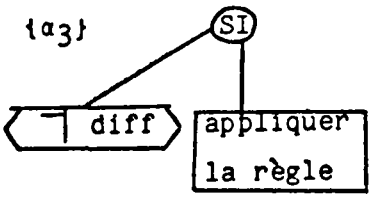
Puisque pour le moment nous considérons que

{i} et (¬fin 1 et ¬diff) [COMPARER] {i} est un résultat,

nous avons terminé la preuve de correction du premier tant que.

Il nous faut encore en faire la preuve d'arrêt :

Nous montrerons plus loin, lors de la preuve de [COMPARER], que chaque application de cette action incrémente p de 1.
Or p est borné par n. Donc après au plus n applications de [COMPARER], on sortira du tant que.
Montrons que :

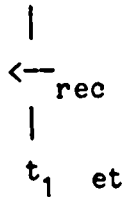


{α3} et ¬(¬diff) => {β} d'après les règles d'effacement de Λ' détaillées precedemment

{α3} et ¬diff [application de la règle] {β} est un résultat

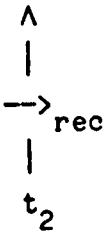
En effet :

{α3} et ¬diff => tous les tests d'égalité ont été des succès, on a la configuration



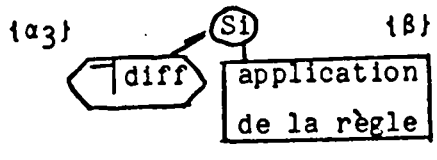
t_1 et t_1 est unifiable avec g.

Après l'utilisation de cette primitive, on obtient la configuration suivante :



t_2

où t_2 est un linéarisé droit de t_3, tel que $t_1 \xrightarrow{(g->d)} t_3$
Ce qui prouve le résultat.
On en conclut donc que :



Prouvons maintenant que :

i et ($\neg$Fin$_1$ et $\neg$diff) | COMPARER | i est un résultat

Montrons tout d'abord que :

i et ($\neg$Fin1 et $\neg$diff) | Marquer le sommet du sous-arbre à comparer | $\{\alpha_4\}$ est un résultat

Où α_4 = { On a la configuration Λ' (avec $t^{(3)}$ en dessous), où $t^{(3)}$ est t" (défini dans i) dans lequel le pointeur $\longrightarrow_{rec}$ est remplacé par un pointeur $\longrightarrow_{rem}$; Un symbole $\overline{\circledast}$ suit le premier fc de $t^{(3)}$ }

Le résultat est clair, il suffit de considérer la description de cette primitive.

Prouvons que :

$\{\alpha_4\}$ | remonter le code au sommet | $\{\alpha_5\}$ est un résultat

avec α_5 = { La première suite dc (α_i) fc de t" relativement à l'ordre sens de parcours est désormais sous le sommet Λ', précédée du pointeur $\longrightarrow_{rem}$. L'endroit où se trouvait cette suite est marqué d'un $\overline{\circledast}$ }

Là encore, les règles définissant cette primitive nous assurent du résultat.

Prouvons que :

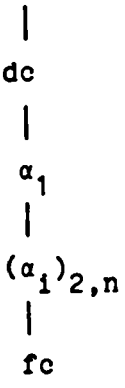
$\{\alpha_5\}$ | chercher le noeud correspondant | $\{\alpha_6\}$ est un résultat

avec

α_6 = { Le pointeur est $\longrightarrow$ch suivi de dc fc, il se trouve au dessus du du noeud de code $\overline{a1 \ldots an}$ }

Chaque règle de cette primitive: $\longrightarrow_{ch}$

1. Fait descendre la suite



par la α^e_i branche du noeud qu'elle surmonte

2. Fait disparaitre le α_1 désormais inutile.

On conduit donc bien le pointeur au noeud de code $\overline{\alpha 1 \ldots . \alpha n}$, en effaçant petit à petit la suite (α_i).

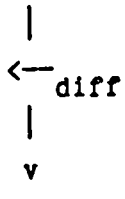
La primitive est donc prouvée.

Montrons maintenant que :

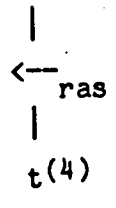
$\{\alpha_6\}$ PREMIERE COMPARAISON $\{\alpha_{10}\}$ est un résultat

avec :

$\alpha_{10} =$ si les sommets des deux sous-arbres à comparer diffèrent, on a la configuration



Sinon on a la configuration



où $t^{(4)}$ est t" dans lequel :

on a effectué le pointeur $\longrightarrow_{rec}$ et la première suite dc | (α_i) | fc

on a marqué de $\overline{\oplus}$ et $\overline{\circledast}$ les sommets des deux sous-arbres à comparer

Nous allons décomposer cette preuve en plusieurs preuves élémentaires :

Montrons que

$\{\alpha_6\}$ [marquer le sommet et charger l'information] $\{\alpha_7\}$ est un résultat

avec :

α_7 = Le noeud de code $\alpha_1 \ldots \alpha_n$ est marqué d'un $\overline{\oplus}$
Le pointeur est père de ce $\overline{\oplus}$, c'est $\longrightarrow_{compl\,f}$, où f est le symbole fonctionnel porté par le noeud de code $\overline{\alpha_1 \ldots \alpha_n}$.

La seule famille de règles de cette primitive entraîne clairement le résultat.

De même :

$\{\alpha_7\}$ [Remonter l'information au sommet] $\{\alpha_8\}$

avec α_8 = on a la configuration $\wedge'$ | $\longleftarrow_{compl\,f}$ | $t^{(4)}$
avec $t^{(4)}$ défini en (α_{10}), $\longrightarrow_{compl\,f}$ en $\{\alpha_7\}$

est clairement un résultat.

Montrons que :

$\{\alpha_8\}$ [descendre jusqu'au sommet de l'arbre à comparer] $\{\alpha_9\}$ est un résultat

avec

α_9 = si les sommets des deux sous-arbres diffèrent, on a la configuration
$\wedge$ | $t^{(5)}$
où $t^{(5)}$ est $t^{(4)}$ dans lequel un pointeur $\longleftarrow_{diff}$ est placé au-dessus du symbole $\overline{\circledast}$

sinon on a la configuration Λ'

$t^{(6)}$

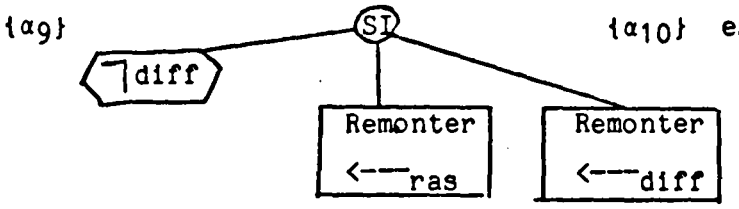
où $t^{(6)}$ est $t^{(5)}$ dans lequel le pointeur $\leftarrow\!\!-\!\!-_{diff}$ est remplacé par $\leftarrow\!\!-\!\!-_{ras}$.

- Par construction de la primitive, le pointeur $-\!\!-\!\!>_{comp'1f}$ (première étape de cette primitive) parcourt l'arbre jusqu'à ce qu'il rencontre un symbole ⊛ .
- Les règles suivantes transforment ce pointeur en :

  $-\!\!-\!\!>_{diff}$ si les sommets diffèrent

  $-\!\!-\!\!>_{ras}$ sinon.

Pour terminer la preuve de [PREMIERE COMPARAISON], montrons que :



est un résultat.

Bien qu'écrite sous forme de décomposition alternative, il s'agit seulement ici de remonter le pointeur créé lors de la phase précédente sous le sommet Λ'.

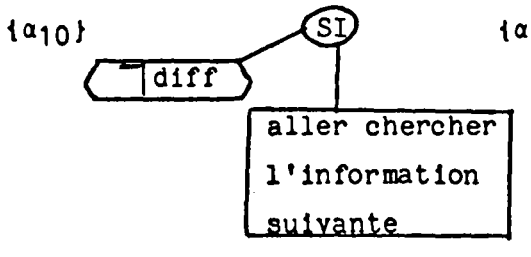
Le résultat est alors immédiat.

Ainsi, nous avons montré que {α9} [PREMIERE COMPARAISON] {α10} est un résultat

Afin d'achever la preuve de COMPARER il nous reste à montrer que :

{α10} [COMPARER LES DEUX SOUS-ARBRES] i est un résultat

Montrons tout d'abord que :

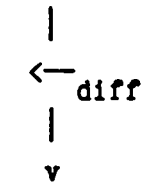


est un résultat.

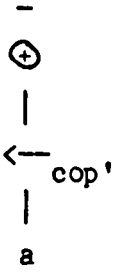
avec :

α_{11} =

- Si les sommets diffèrent, on a la configuration



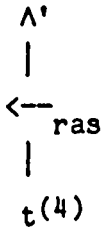
- Sinon on a la configuration Λ' où $t^{(7)}$ est $t^{(4)}$ dans lequel le pointeur est $\longrightarrow_{copg}$, placé au-dessus du premier fils du noeud marqué d'un $\overline{\oplus}$, g étant le symbole correspondant à ce noeud.
- Si le descendant du noeud marqué par $\overline{\oplus}$ est une constante a, on a la configuration



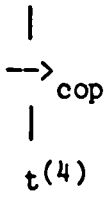
Preuve :

a. Si les sommets diffèrent, la primitive n'est pas appliquée, le résultat est vrai.

b. Sinon, on a obtenu, lors de la phase précédente, la configuration

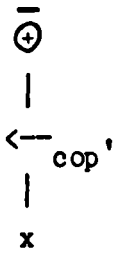


qui engendre ensuite



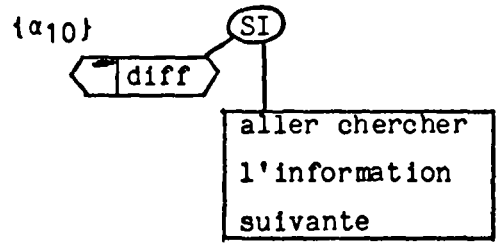
Ce pointeur $\longrightarrow_{cop}$ va rechercher le symbole $\overline{\oplus}$. Il se transforme alors en $\longrightarrow_{cop'}$, recherchant dans le sous-arbre de $\overline{\oplus}$ le premier noeud non marqué d'un $\oplus$. Il chargera alors l'information correspondante, se transformant alors ne $\longrightarrow_{copg}$, si g est le symbole porté par ce noeud.

c. Si un tel noeud n'existe pas (i.e : tous les noeuds ont déjà été marqués), on obtient la configuration



qui correspond à la condition d'arrêt Fini.

Ceci nous permet donc d'affirmer que



{α11} est un résultat

Prouvons le tant que :

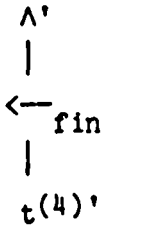
On définit son invariant i_1 de la manière suivante :

$i_1 =$ n_1 est le nombre de noeuds du modèle, p_1 le nombre de ces noeuds déjà comparés.

Si le $(p_1+1)^e$ noeud du sous-arbre à comparer diffère de son modèle, ou a la configuration

Sinon, ce noeud est marqué par ⊛ , le pointeur est <---copg, situé au-dessus du premier noeud non marqué du modèle (g étant le symbole porté par ce noeud).

Si $p_1=n_1$, on a la configuration



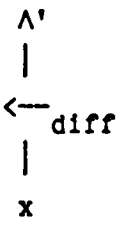
où $t^{(4)'}$ est $t^{(4)}$ dans lequel les descendants des noeuds marqués ⊖ et ⊛ sont tous marqués respectivement par les symboles ⊖ et ⊛.

On a clairement :

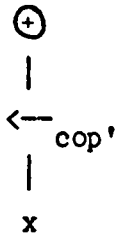
α_{11} et ($\neg$fin et diff) => i_1

Soit α_{12} défini de la façon suivante :

α_{12} = si les deux sous-arbres en cours de comparaison diffèrent, on a la configuration



Sinon on a la configuration



et un nouveau couple de sous-arbres a été comparé avec succès (p <-- p+1).

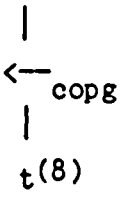
On a α_1 et (fin ou diff) => α_{12}

Il nous reste à montrer que i_1 est invariant :

Tout d'abord, montrons que :

i_1 et ($\neg$fin et $\neg$diff) | Remonter l'information | $\{\alpha_{13}\}$ est un résultat, avec :

α_{13} = On a la configuration



où $t^{(8)}$ est $t^{(4)}$ dans lequel les (p_1+1) premiers noeuds descendants du sommet marqué $\overline{\ominus}$ sont marqués d'un $\ominus$, les p_1 premiers noeuds descendants du sommet marqué $\overline{\circledast}$ sont marqués d'un $\circledast$.

g est le symbole du dernier noeud descendant de $\overline{\ominus}$ qui soit marqué d'un $\ominus$

D'après le rôle joué par la primitive [Remonter l'information], ceci est clairement un résultat.

Montrons que :

$\{\alpha_{13}\}$ [Redescendre jusqu'au sommet de l'arbre à comparer] $\{\alpha_{14}\}$ est un résultat

Ceci est clair si l'on pose :

α_{14} = { p_1 noeuds ont été comparés avec succès, (p_1+1) noeuds du modèle sont marqués. Le pointeur est $\longrightarrow_{comp2g}$, père de $\overline{\circledast}$, g est le symbole du dernier noeud du modèle marqué d'un $\oplus$. }

Prouvons que :

$\{\alpha_{14}\}$ [Rechercher le noeud à comparer avec l'information] $\{\alpha_{15}\}$ est un résultat

Là encore, c'est clair si l'on pose :

α_{15} = { p_1 noeuds ont été comparés avec succès, (p_1+1) noeuds du modèle sont marqués. Le pointeur est $\longrightarrow_{comp'2g}$, père du $(p_1+1)^e$ noeud de l'arbre à comparer. }

Enfin, pour terminer la preuve de ce tant que, il nous reste à montrer que

$\{\alpha_{15}\}$ si $\neg$diff alors
Marquer ce noeud ;
Remonter à vide ;
Aller chercher l'information suivante ;
Sinon Remonter indicateur de différence ;
finsi $\{i_1\}$

a. $\{\alpha_{15}\}$ et (diff) [Remonter indicateur de différence] $\{i_1\}$ est un résultat

En effet, les deux symboles à comparer n'étant pas égaux, le pointeur de vient $\longrightarrow_{diff}$ qui est sous le sommet Λ' à la fin de cette primitive.

b : $\{\alpha_{15}\}$ et (diff) [Marquer ce noeud] $\{\alpha_{16}\}$ est un résultat, si l'on pose :

D'après le rôle joué par la primitive | Remonter l'information |, ceci est clairement un résultat.

Montrons que :

$\{\alpha_{13}\}$ | Redescendre jusqu'au sommet de l'arbre à comparer | $\{\alpha_{14}\}$ est un résultat

Ceci est clair si l'on pose :

α_{14} = { p_1 noeuds ont été comparés avec succès, (p_1+1) noeuds du modèle sont marqués. Le pointeur est $\longrightarrow_{comp2g}$, père de $\overline{\circledast}$, g est le symbole du dernier noeud du modèle marqué d'un $\oplus$. }

Prouvons que :

$\{\alpha_{14}\}$ = | Rechercher le noeud à comparer avec l'information | $\{\alpha_{15}\}$ est un résultat

Là encore, c'est clair si l'on pose :

α_{15} = { p_1 noeuds ont été comparés avec succès, (p_1+1) noeuds du modèle sont marqués. Le pointeur est $\longrightarrow_{comp'2g}$, père du $(p_1+1)^e$ noeud de l'arbre à comparer. }

Enfin, pour terminer la preuve de ce tant que, il nous reste à montrer que

$\{\alpha_{15}\}$ si $\neg$diff alors
Marquer ce noeud ;
Remonter à vide ;
Aller chercher l'information suivante ;
Sinon Remonter indicateur de différence ;
finsi $\{i_1\}$

a. $\{\alpha_{15}\}$ et (diff) | Remonter indicateur de différence | $\{i_1\}$ est un résultat

En effet, les deux symboles à comparer n'étant pas égaux, le pointeur devient $\longrightarrow_{diff}$ qui est sous le sommet Λ' à la fin de cette primitive.

b : $\{\alpha_{15}\}$ et (diff) | Marquer ce noeud | $\{\alpha_{16}\}$ est un résultat, si l'on pose :

α_{16} = $p_1 \leftarrow p_1 + 1$

Les p_1 premiers noeuds de chaque sous-arbre ont été comparés avec succès et sont marqués, suivant le cas, d'un ⊕ ou d'un ⊛

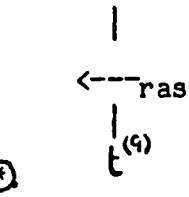
Le pointeur est $\longrightarrow_{ras}$, situé au-dessus du ⊛ marquant le $p_1^{ème}$ noeud du sous-arbre à comparer.

Remarquons au passage, qu'a lieu ici l'incrementation de p_1, ce qui permet d'affirmer que le tant que s'arrêtera, car $p_1 \leq n_1$.

Si l'on pose :

α_{17} = p_1 noeuds de chaque sous-arbre ont été comparés avec succès, les p_1 premiers noeuds de chaque sous-arbre sont marqués par + ou *.

On a la configuration



où $t^{(g)}$ est t dans lequel les p_1 premiers descendants des sommets marqués ⊕ et ⊛ sont marqués respectivement par ⊕ et ⊛

On a alors clairement le résultat suivant :

{α_{17}} | aller chercher information suivante | {i_1} il suffit d'utiliser la preuve de cette primitive

Ceci termine donc la preuve du tant que (celui-ci s'arrêtant soit en cas de différence, soit lorsque $p_1 > n_1$).

Il nous faut encore montrer que :

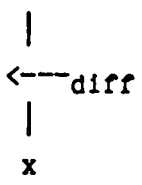
{α_{12}} si ¬diff alors
Effacement partiel ;
Regenerer un pointeur de recherche ;
Chercher un arbre à comparer ;
Sinon effacement total ;
finsi {i}

est un resultat

a. {α_{12}} et (diff) | effacement total | i est un résultat.

En effet, le dernier couple testé (arbre/modèle) empêche l'unification de t et de $\bar{g}$.

La configuration



entraîne alors la génération d'un pointeur d'effacement total, supprimant sous Λ tous les symboles auxiliaires (ceux n'appartenant pas à Σ)
On retrouve donc l'arbre duquel on était parti.

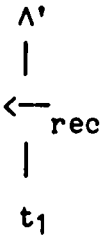
$\{\alpha_{12}\}$ et $\{$ diff$\}$ effacement partiel ;
Regenerer un pointeur de recherche ;
Chercher un arbre à comparer ;
$\{i\}$
est un résultat.

En effet :
Le dernier test d'égalité a été un succès, donc :

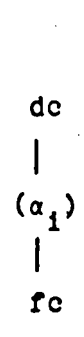
1. On efface les symboles de l'ensemble $\{\bar{\oplus}, \bar{\circledast}, \oplus, \circledast\}$
2. On recherche un arbre à comparer en lançant un pointeur de recherche

\

cas a : S'il n'existe plus de tels arbres, on a la configuration



cas b : Sinon on a la configuration Λ' où t' est t dans lequel les p premières séquences dc ont été effacées et où un pointeur



$\longrightarrow_{rec}$ s'est intercalé devant le $(p+1)^e$ dc.

Nous terminons ainsi la preuve de l'algorithme de linéarisation à gauche d'une règle.

CHAPITRE V

Simulation linéaire
d'un système de réécriture

Simulation d'une règle générale d'un système de réécriture

Les trois chapitres précédents nous permettent de simuler linéairement des règles de réécriture non linéaires à droite ou à gauche.

Nous montrons maintenant comment on peut simuler une règle quelconque (linéaire ni à droite, ni à gauche).

Pour cela remarquons que même si une règle est linéaire, on peut lui appliquer les simulations décrites précédemment.

En effet, simuler une règle linéaire revient à laisser inchangés ses membres gauche et droit. Le pointeur $\longrightarrow_{rec}$ parcourant chacun de ces arbres remontera donc directement sous le sommet, on obtient la condition d'arrêt de la simulation.

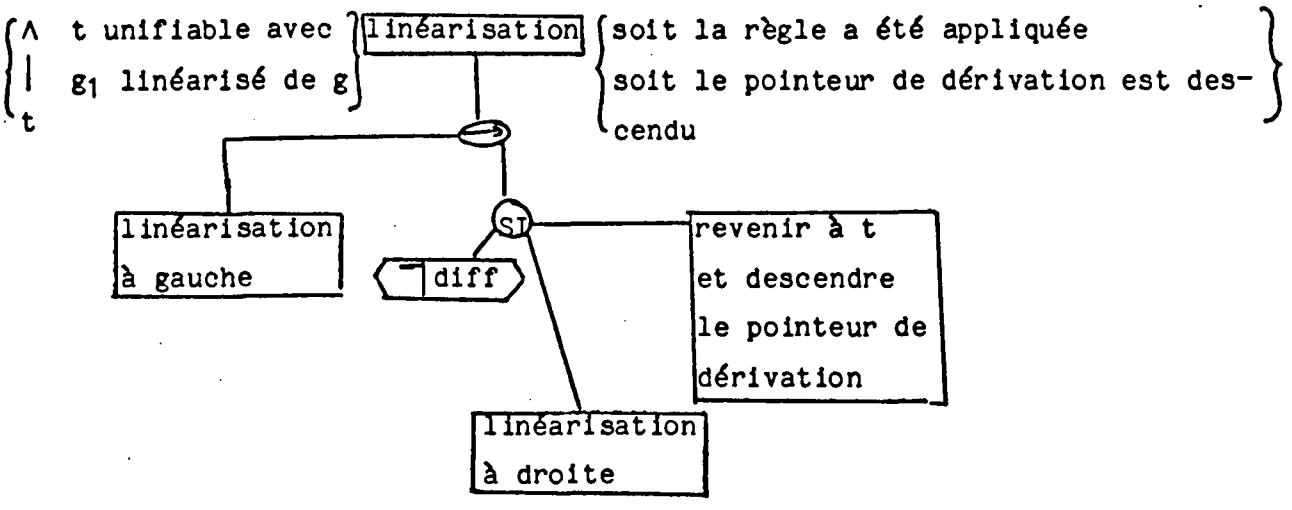
Nous allons donc :

- Appliquer les simulations (à droite comme à gauche) à toutes les règles (même linéaires) du système de départ.
- Définir un algorithme général de linéarisation (à droite et à gauche).

Considérons une règle de réécriture quelconque g —> d définie sur $T(\Sigma)$, ni g ni d n'étant linéaires.

Notre but est de simuler linéairement cette règle ; il nous faudra d'abord réduire la non-linéarité à gauche pour nous occuper ensuite de la non-linéarité à droite.

Ce que l'on peut exprimer de la façon suivante :



La seule partie qu'il nous reste à décrire est le passage de la simulation gauche à la simulation droite.

Il s'agit donc de la primitive [appliquer la règle] dont nous avions repoussé la description

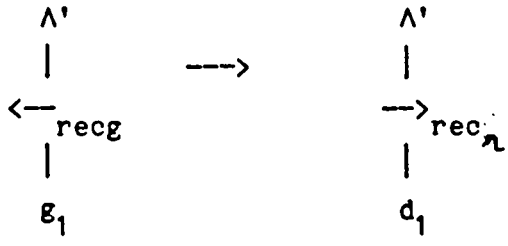
Nous la définissons de la manière suivante :

Soit $g \longrightarrow d$ la règle de R, système de départ

g_1 linéarisé gauche de g

d_1 un linéarisé droit indicé de d.

On crée les règles :



On peut remarquer que pour éviter les confusions, il est nécessaire de différencier les pointeurs $\longrightarrow_{rec}$ selon qu'ils agissent dans la simulation gauche ou droite.

Plus généralement, cela sera fait pour tous les pointeurs. Cela ne change aucun des résultats précédemment obtenus.

La définition de [appliquer la règle],

ainsi que les précautions prises quant au nom des pointeurs nous permettent donc d'affirmer que toute règle de réécriture est simulable par une suite finie de règles de réécriture linéaires.

Généralisons ce résultat, en montrant que tout système de réécriture est simulable, sur les termes clos, par un système de réécriture linéaire.

Soit $R = \{(g_i \longrightarrow d_i)\ i \in [1,n]\}$ un système de réécriture sur $T(\Sigma)$.

Nous avons montré, dans les chapitres précédents, que pour chaque règle $g_i \longrightarrow d_i$, il existe un système de réécriture R_i sur $\Sigma' \supset \Sigma$ tel que :

$$\forall\, t, u \in T(\Sigma)^1_0 \quad t \xrightarrow[t/\epsilon]{i} u \iff \Lambda \xrightarrow[R_i]{*} u \text{ (en } t\text{)}$$

Appelons R_0 le système de réécriture (linéaire) correspondant aux structures de contrôle.

Théorème :

$R_\ell = \bigcup_{i=o}^{n} R_i$ simule le système de réécriture R

Démonstration :

a. Soient $(t,u) \in T(\Sigma)^1_0$ $\quad t \xrightarrow[R]{*} u$

Pour chaque règle de R impliquée dans cette dérivation, il existe une suite de dérivations dans R_ℓ la simulant (il suffit pour cela d'utiliser le pointeur $\longrightarrow_{fac}$ qui permettra d'engendrer la règle voulue à l'endroit voulu).

Il suffit donc d'enchaîner autant de parcours de $\longrightarrow_{fac}$ que de règles de R à appliquer.

b. réciproquement :

Soient $(t,u) \in T(\Sigma)^1_0$ tels que

$$\begin{array}{ccc} SO' & \xrightarrow[R_\ell]{*} & SO' \\ | & & | \\ \longrightarrow_1 & & \longrightarrow_2 \\ | & & | \\ t & & u \end{array}$$

avec $\longrightarrow_1, \longrightarrow_2 \in \{\longrightarrow_{fac}, \longrightarrow_{ob}\}$

Montrons qu'alors $t \xrightarrow[R]{*} u$

- Si t = u, alors $t \xrightarrow[R]{o} u$

- Si t ≠ u le pointeur $\longrightarrow_1$ a donc parcouru t et s'est transformé en Λ à une certaine occurence de t. (Le pointeur seul ne modifiant pas t).
Or ceci n'est possible que lorsqu'une règle de R est applicable à cette occurence. La suite des dérivations qui commence alors sert à simuler linéairement une règle de R, jusqu'à la régénération du pointeur de contrôle.

donc $t \xrightarrow[R]{*} u$

Considérons la structure de contrôle qui consiste à marquer les sommets.
Nous avons alors les résultats suivants :

Soit R système de réécriture sur $T(\Sigma)$, $R_\ell = R_o + \bigcup_{i=1}^{n} R_i$, R_o relatif au marquage des sommets.

P_1 : R est noetherien ssi R_ℓ est noetherien à partir de $T(\Sigma)^1_o$

P_2 : R est confluent ssi R_ℓ est confluent à partir de $T(\Sigma)^1_o$

P_3 : R est acyclique ssi R_ℓ est acyclique à partir de $T(\Sigma)^1_o$

Ces propriétés traduisent le fait que R_ℓ conserve les propriétés globales de R.

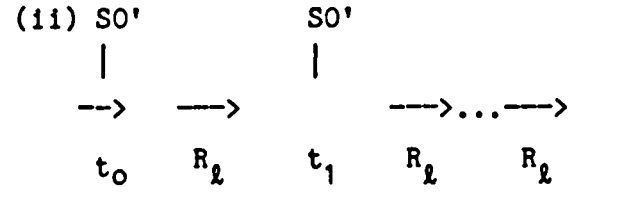
Démonstration de P_1 :

Supposons R_ℓ non noetherien à partir de $T(\Sigma)^1_o$

Il existe donc une suite infinie $(t_i)_{i\in N}$ telle que :

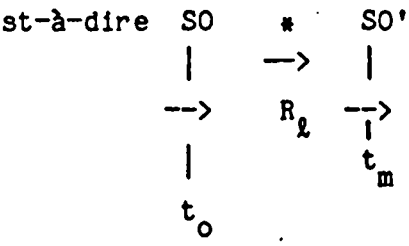
(i) $t_i \in T(\Sigma')^1_0$ pour $i > 0$, $t_o \in T(\Sigma)^1_0$ $\forall\, i \neq j \quad t_i \neq t_j$

(ii)



Par construction de R_ℓ, et puisque $t_o \in T(\Sigma)^1_o$, $\exists\ m \in N$ / $t_m \in T(\Sigma)^1_o$

C'est-à-dire



$t_o \neq t_m$ donc $t_o \xrightarrow[R]{+} t_m$

On peut donc construire une suite infinie de dérivations dans R, qui n'est donc pas noetherien.

Réciproquement :

Si R n'est pas noetherien, alors R_ℓ ne l'est pas non plus, puisque R_ℓ simule R.

Démonstration de P2 :

a. Soit R confluent.

En particulier, nous avons pour les termes clos :

$$\forall\, t \in T(\Sigma)^1_0 \quad \left.\begin{array}{l} t \xrightarrow[R]{*} t_1 \\ t \xrightarrow[R]{*} t_2 \end{array}\right\} \Rightarrow \exists\, t' \in T(\Sigma) \text{ tel que } \left\{\begin{array}{l} t_1 \xrightarrow[R]{*} t' \\ t_2 \xrightarrow[R]{*} t' \end{array}\right.$$

(Remarquons que $t_1, t_2, t' \in T(\Sigma)^1_o$).

Par définition de la simulation de R par R_ℓ, nous pouvons écrire :

$$(1)\ \forall\, t \in T(\Sigma)^1_o \quad \left.\begin{array}{l} SO' \xrightarrow{*} SO' \\ | \quad \longrightarrow \quad | \\ \longrightarrow \ R_\ell \ \longrightarrow \\ t \qquad t_1 \\ SO' \xrightarrow{*} SO' \\ | \quad \longrightarrow \quad | \\ \longrightarrow \ R_\ell \ \longrightarrow \\ | \qquad | \\ t \qquad t_2 \end{array}\right\} \Rightarrow \exists\, t' \in T(\Sigma)^1_o \ t_q \left\{\begin{array}{l} SO' \xrightarrow{*} SO' \\ | \quad \longrightarrow \quad | \\ \longrightarrow \ R_\ell \ \longrightarrow \\ t_1 \qquad t' \\ SO' \xrightarrow{*} SO' \\ | \quad \longrightarrow \quad | \\ \longrightarrow \ R_\ell \ \longrightarrow \\ | \qquad | \\ t_2 \qquad t' \end{array}\right.$$

Ce qui n'est autre que l'expression de la confluence de R_ℓ à partir de $T(\Sigma)^1_o$

b. Réciproquement :

Déf : Soit $t \in T(\Sigma')^1_o$. On dira que t est "bien construit" s'il se déduit d'un arbre $t_o \in T(\Sigma)^1_o$ par applications de règles (ou d'initialisations) de R_ℓ. On peut alors donner une expression de la confluence de R_ℓ à partir de $T(\Sigma)^1_0$

$$(2)\begin{cases} R' \text{ confluent} \\ \text{à partir de } T(\Sigma)^1_0 \end{cases} \Leftrightarrow \forall\, t \text{ "bien construit"} : t \in T(\Sigma')^1_0 \Rightarrow \left[\begin{array}{c} SO' \;\; * \;\; SO' \\ | \;\; \longrightarrow \;\; | \\ t \;\; R_\ell \;\; t_1 \end{array} \right] \Rightarrow \exists\, t' \in T(\Sigma')^1_0$$

$$\left[\begin{array}{c} SO' \;\; * \;\; SO' \\ | \;\; \longrightarrow \;\; | \\ t \;\; R_\ell \;\; t \end{array} \right]$$

tel que

$$\begin{cases} SO' \;\; * \;\; SO' \\ | \;\; \longrightarrow \;\; | \\ t_1 \;\; R_\ell \;\; t' \\ SO' \;\; * \;\; SO' \\ | \;\; \longrightarrow \;\; | \\ t_2 \;\; R_\ell \;\; t' \end{cases}$$

Remarque :

Par définition, t "bien construit" => t_1, t_2, t' "bien construits"

Lemme :

Soit $t \in T(\Sigma')^1_0$ "bien construit". Alors il existe u, $v \in T(\Sigma)^1_0$ tels que

$$\begin{array}{ccccccc} SO' & * & SO' & * & SO' \\ | & \longrightarrow & | & \longrightarrow & | \\ \longrightarrow & R_\ell & t & R_\ell & \longrightarrow \\ | & & & & | \\ u & & & & v \end{array}$$

Démonstration :

1. Par définition d'un arbre "bien construit", u existe.
2. t est donc une phase de la simulation d'une règle de R appliquée à u. Donc il existe $v \in T(\Sigma)^1_0$ tel que $u \xrightarrow[R]{*} v$

Ce qui termine la démonstration du lemme.

Grâce à ce lemme, nous voyons que l'expression (2) est équivalente à l'expression (1).

On obtient donc à partir de là l'expression de la confluence de R, ce qui termine la démonstration de la proposition P2.

Démonstration de P3 :

a. Si R possède un cycle, alors R_ℓ en a un aussi, car il simule R.

b. Réciproquement : Si R_ℓ possède un cycle, ça ne peut être ni dans les structures de contrôle (on est toujours dans le cas où le contrôle se fait par marquage du sommet), ni dans le cours d'une simulation (par construction). Ce cycle est donc induit par R.

Nous voyons donc que lors de la simulation linéaire, les principales propriétés globales du système de réécriture simulé sont conservées.

Par contre, en général, les propriétés locales de R ne sont pas conservées. En effet, toute propriété locale de R correspond à une seule application de règle de R, règle simulée dans R_ℓ par une suite de règles.

Réciproquement, R_ℓ peut posséder des propriétés locales que n'a pas le système de départ R. (On peut le comprendre en considérant que R_ℓ est "plus fin" que R).

Par conséquent, les seules généralisations aux cas de systèmes non linéaires de résultats concernant les systèmes de réécriture linéaires seront celles ne faisant pas intervenir les propriétés locales des systèmes considérés.

CHAPITRE VI

Application aux théories Equationnelles

Application aux théories équationnelles

Introduction :

Nous n'avons jusqu'à présent considéré que la simulation des systèmes de réécriture. C'est-à-dire qu'il importait peu que les règles engendrées pour ces simulations, du type t —>u, soient utilisables dans l'autre sens u —>t.

Nous nous intéresserons dans ce chapitre aux théories équationnelles, c'est-à-dire à des types d'objets définis par un ensemble d'égalité entre termes. Nous les considérerons comme des systèmes de réécriture reversibles. Plus clairement, à toute équation du type $t_i = t'_i$ nous associerons deux règles de réécriture :

$$t_i \longrightarrow t'_i \qquad t'_i \longrightarrow t_i$$

Pour que ceci ait un sens, nous devrons limiter notre étude au cas des théories équationnelles dont chaque équation est complète, c'est-à-dire telle que $V(t_i) = V(t'_i)$.

Notre démarche va être la suivante :

1. A un système d'équations $E = \{u_i = v_i / i \in I\}$, on associe le système de réécriture $R_E = \{u_i \longrightarrow v_i, v_i \longrightarrow u_i / u_i = v_i \in E\}$.

   (Ce sdr sera dit <u>symétrique</u> car si $t \longrightarrow u \in R_E$, alors $u \longrightarrow t \in R_E$).

2. Nous simulons, de la façon détaillée dans les chapitres précédents, "la moitié de R_E" par un sdr R' dont les règles seront <u>réversibles</u> (informellement, les règles sont applicables sans ambiguïté dans l'autre sens).

3. Nous définissons $R^= = R' \cup R'^{-1}$, sdr symétrique simulant R_E.

   Ceci nous permet en retour d'associer à $R^=$ un système d'équations E' qui simulera E.

   PLus précisemment, nous posons les définitions suivantes :

<u>Déf 1</u> : Soit $\Sigma' \supset \Sigma$

On dira que E', (système d'équations défini sur Σ'), simule E (défini sur Σ)

<u>ssi</u> : $\forall t \in T(\Sigma)^1_0$

$$\text{Classe}_E(t) = \text{Classe}_{E'}(t) \cap T(\Sigma)^1_0$$

<u>Déf 2</u> : Un sdr R sera dit symétrique ssi :

$u \longrightarrow v \in R \iff v \longrightarrow u \in R$

<u>Déf 3</u> : Soit R sdr

On définit l'ensemble $R^{-1} = \{ d \longrightarrow g / g \longrightarrow d \in R\}$

<u>Remarque</u> : Si, pour toute règle de R, $V(g) = V(d)$ alors R^{-1} est un sdr.

On peut alors parler du symétrisé de R $R^= = R \cup R^{-1}$

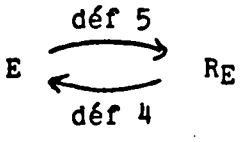
<u>Déf 4</u> : A tout sdr symétrique R, on peut associer un ensemble d'équations E_S tel que :

$$E_S = \{ u = v / u \longrightarrow v \in R\}$$

<u>Déf 5</u> : A tout ensemble d'équations E, on peut associer un sdr R_E symétrique tel que :

$$R_E = \{ u \longrightarrow v, v \longrightarrow u / u = v \in E\}$$

On obtient donc le schéma suivant :



Lemme :

Si R_E est simulé fortement par S, sdr symétrique, alors E est simulé par E_S.

Preuve :

$$t \equiv_E t' \iff (t \xrightarrow[R_E]{*} t' \text{ et } t' \xrightarrow[R_E]{*} t)$$

$$\Rightarrow (t \xrightarrow[S]{*} t' \text{ et } t' \xrightarrow[S]{*} t)$$

$$\Rightarrow t \equiv_{ES} t'$$

(E_S existe car S est symétrique).

Avant de détailler ces résultats et les moyens d'y arriver, nous allons décrire un exemple de simulation d'un ensemble d'équations par un ensemble d'équations linéaire.

Avertissement :

Pour restreindre le nombre de règles à décrire, nous allons considérer qu'à x, y, z sont nécessairement substitués des termes de la forme $s^n(0)$.

(Cela revient à se donner des sortes sur nos opérateurs).

Soit E = { a = b(x, x) } Σ = { a, b, s, 0 }

On définit :

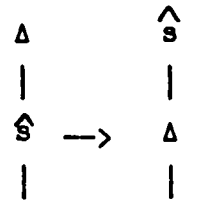
R' =

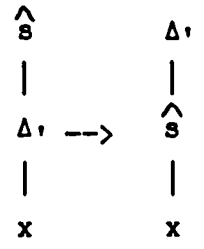
(1) a(x) → b(x, Δ)

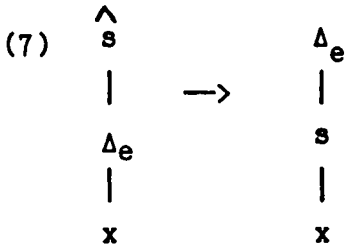
(2) b(b(x, Δ), Δ) → b(x, 0)

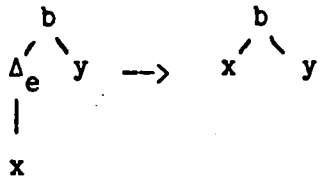
(5) b(b(Δ, y), s) → b(Δ(x), s(y))

(6) Δ(0) → Δ_e(0)

(3) 

(4) 

(7) 

(8) 

R' est défini de telle façon qu'on recopie, comme fils droit de b , la suite s^n figurant à gauche.

Δ est un pointeur allant chercher, à gauche, le s non marqué le plus haut. Δ', engendré lors de cette rencontre remonte sous le sommet puis crée un s à droite.

Lorsque Δ rencontre le '0' de gauche, il devient Δ_e, pointeur d'effacement otant toutes les marques $\hat{s}$ (présentes uniquement à droite), puis le marquage au sommet.

On définit alors $S = R' \cup R'^{-1}$, et E_S, obtenu en remplaçant les règles de R' par des égalités, simule linéairement E.

Rappelons cependant que cet exemple est très simplifié : généralement, la définition de R' doit tenir compte du fait que les variables peuvent être des termes quelconques, et d'autre part contrôler la simulation.

Dans les pages suivantes, nous revenons plus formellement, et parfois en utilisant des notions et des méthodes sensiblement différentes, sur ces simulations.

Simulation forte, simulation faible :

Nous avons défini dans les chapitres précédents deux façons de simuler linéairement une règle de réécriture.L'une d'elle nécessitait le marquage du sommet des termes, contrainte très forte, tandis que l'autre, gelant les sous-arbres du terme à dériver, nécesitait un nouvel alphabet de symboles barrés.

Ces deux structures de contrôle, aux propriétés différentes correspondent chacune à l'une des définitions de la simulation que nous introduisons ici :

Définition 1 :

Soient R, S deux systèmes de réécriture définis respectivement sur $T(\Sigma)$ et $T(\Sigma')$ tels que $\Sigma \subset \Sigma'$.

On dit que S simule fortement R si :

$\forall (u,v) \in T(\Sigma')^2$ 1. $u \xrightarrow[R]{*} v \Rightarrow u \xrightarrow[S]{*} v$

2. Si $u \in T(\Sigma)$ $u \xrightarrow[S]{*} v \Rightarrow \exists v_1 \in T(\Sigma)$

tel que $u \xrightarrow[S]{*} v_1 \xrightarrow[S]{*} v$

et $u \xrightarrow[R]{*} v_1$

Définition 2 :

Sous les mêmes hypothèses on dira que S simule faiblement R si :

$\exists f : T(\Sigma') \longrightarrow T(\Sigma')$ tel que :

$$u \xrightarrow[R]{*} v \Rightarrow f(u) \xrightarrow[S]{*} f(v)$$

et

$$\left.\begin{array}{l} f(u) \xrightarrow[S]{*} f(v) \\ \text{et } u \in T(\Sigma) \end{array}\right\} \Rightarrow \begin{array}{l} \exists v_1 \in T(\Sigma) \text{ tel que } f(u) \xrightarrow[S]{*} f(v_1) \xrightarrow[S]{*} f(v) \\ \text{et } u \xrightarrow[R]{*} v_1 \end{array}$$

Remarque :

Le marquage au sommet correspond à la fonction f. La structure de contrôle qui lui est associé conduit donc à une simulation faible.

Par contre, le gel des sous-arbres, si l'on s'autorise des étapes de dégel à la fin des transformations relève de la simulation forte.

Ces deux types de simulation ont des propriétés fort différentes, qui expliquent que, suivant les propriétés que l'on désire conserver, l'on choisisse l'une ou l'autre.

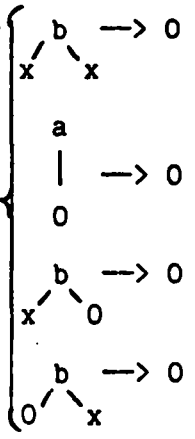
En particulier, nous avons montré que la simulation faible que nous avons employé auparavant conservait la propriété de terminaison finie. Il n'en est pas de même pour la simulation forte :

Lemme : -

La simulation forte ne conserve pas la terminaison finie.

Montrons-le sur un exemple :

Soit Σ = { b , a, $\bar{a}$, 0} et R défini par



On voit que pour $t \in T(\Sigma)$:

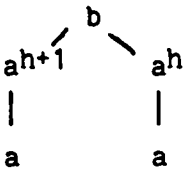
$t \xrightarrow[R]{*} 0$ si un b a deux fils jumeaux ou si $\#(0,t) \neq 0$

$t \xrightarrow[R]{0} t$ sinon

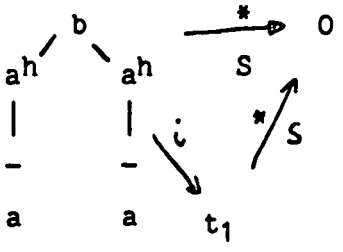
R est noetherien. Soit S simulant R et h > sup { hauteur des membres gauches de règle de S }

Puisque S a un nombre de règles fini, ce sup existe et est fini.

Alors il existe une règle de S applicable à t_0 =



En effet :



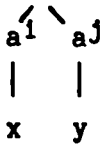
i étant une règle de S : $g_i \dashrightarrow d_i$

Deux cas se présentent :

1. $g = a^j \dashrightarrow d$ avec $j < h$. La règle est donc applicable à t_0

(même conclusion si x est remplacé par $\bar{a}$)

2. g =



i, j < h

la règle est ici encore applicable à t_0

Il existe donc une règle de S applicable à t_o. On a donc le schéma suivant :

$t_o \underset{g_i \to d_i}{|\!---}\ t' \overset{*}{|\!---}_S t_o$, puisque S simule R et $t_o \underset{o}{|\!---} t_o$ dans R.

S appliqué à t_o possède donc un cycle, il n'est donc pas noetherien. Remarquons cependant que ce cycle est induit par la présence d'une non-linéarité gauche dans le système de départ. (La simulation de la non linéarité à droite conserve l'aspect noetherien).

Reversibilité :

Afin de pouvoir simuler une règle g —> d et son inverse d —> g, dans le but de simuler des théories équationnelles, nous affinons la définition de la simulation utilisée dans les chapitres précédents de la façon suivante :

a. Tous les symboles auxiliaires (marquages et pointeurs) sont désormais indicés par le numéro de la règle de départ qu'ils simulent.

b. On n'utilise plus, pour repérer les couples de termes à comparer (ou à copier), les suites

dc
|
α_i
|
fc

mais on force de façon déterministe (enchaînement de règles) les pointeurs à aller de l'un à l'autre, ce qui peut être fait finiment pour chaque règle

Les autres constructions (par exemple, l'effacement des ⊕) ont déjà été définies de telle façon qu'elles puissent être utilisées sans ambiguïté dans le sens d —> g. D'autre part, nous supposerons dans tout ce qui suit que la structure de contrôle de simulation employée est celle qui consiste à geler les sous-arbres.

Parenthesage d'une dérivation :

Lorsqu'on simule fortement une dérivation du type $t \xrightarrow{*} u$, on simule séparément ou parallélement un certain nombre de règles du système de départ.

Pour chacune de ces règles, la démarche est la suivante :

- Marquage du point d'application de la règle
- Gel du sous-arbre correspondant

- simulation proprement dite
- dégel et effacement des symboles auxiliaires

On peut décrire cette simulation de la manière suivante :

- On fait correspondre à chaque début de simulation de règle une "ouverture de parenthèse" $(_{i,u}$ signifiant qu'on commence à simuler la règle i à l'occurence u de l'arbre.
- De même on associe une parenthèse fermante $)_{i,u}$ à la fin de la simulation de cette règle.

On obtient de cette façon une "trace" de la simulation composée d'une suite de parenthèses telle que pour chaque parenthèse $(_{i,u}$ appartenant à la suite, $)_{i,u}$ y appartient aussi.

(On aura autant de couples de parenthèses correspondantes que de règles du système de départ utilisées dans $t \xrightarrow{*} u$)

Nous définissons la notion de bon parenthèsage :

Définition :

La trace d'une simulation est bien parenthèsée ssi

- Tout couple $(_{i,u})_{i,u}$ est tel que s'il contient une parenthèse $(_{j,\omega}$ ou $)_{j,\omega}$, il contient la parenthèse associée.
- La parenthèse ouvrante précède toujours la parenthèse fermante correspondante

Proposition :

Soient R,S deux systèmes de réécriture, Σ et Σ' les alphabets gradués associés, et S simule R.

Soient t_1, $t_2 \in T(\Sigma)$

Si $t_1 \xrightarrow[S]{*\delta} t_2$ alors la trace de δ peut s'écrire de façon bien parenthèsée.

Démonstration :

Puisque t_1, $t_2 \in T(\Sigma)$ et S simule R, il existe une dérivation δ' telle que $t_1 \xrightarrow[R]{+\delta'} t_2$

Donc :

$$t_1 \xrightarrow[R]{i_1} t'_1 \xrightarrow[R]{i_2} t'_2 \ldots \xrightarrow[R]{i_n} t'_n \xrightarrow[R]{i_{n+1}} t_2$$

et puisque S simule R :

$$t_1 \xrightarrow[(\ i_1,u_1]{S} t''_1 \xrightarrow[S]{*} t'''_1 \xrightarrow[)\ i_1,u_1]{S} t'_2 \ \ldots\ldots\ \xrightarrow[)\ i_{n+1},\omega]{S} t_2$$

La trace de δ peut donc s'écrire de manière très bien parenthesée (chaque couple est isolé des autres), ce qui termine la preuve de la proposition.
Lors de la simulation d'une dérivation, on peut donc toujours s'arranger pour ne simuler qu'une règles à la fois, la simulation "anarchique" pouvant toujours s'y ramener.
Ayant défini les notions nécessaires pour la suite du chapitre, nous pouvons énoncer trois lemmes, après avoir défini l'inverse d'un système de réécriture :

Définition : Soit $R = \{(g_i \dashrightarrow d_i)\ i \in [1,N]\}$ un système de réécriture.
On appelle R^{-1} l'ensemble défini par :

$$R^{-1} = \{(d_i \dashrightarrow g_i) \ / \ (g_i \dashrightarrow d_i) \in R\}$$

Remarquons que R^{-1} n'est un système de réécriture que si $V(g_i) = V(d_i)$ pour tout i (c'est-à-dire si R est complet)

Soient R, R' deux systèmes de réécriture (linéaires à droite) sur $T(\Sigma)$ complets

S, S' deux systèmes de réécriture linéaires sur $T(\Sigma') \supset T(\Sigma)$

Lemme 1 : S simule R ==> S^{-1} simule R^{-1}

lemme 2 : S simule R ==> S u S^{-1} simule R u R^{-1}

lemme 3 : S simule R, et S' simule R', et les alphabets des symboles auxiliaires de S et S' sont disjoints ==> S u S' simule R u R'

Démonstration du lemme 1 :

Soit $t \xrightarrow[i,\omega]{} u$ une transformation dans R, correspondant à l'application de la règle i à l'occurence ω de t.
Puisque S simule R, la transformation suivante est définie de façon déterministe :

$$t \xrightarrow[(i,\omega]{} t_1 \xrightarrow[S]{i^*} u_1 \xrightarrow[)i,\omega]{} u$$

à $t \xrightarrow[i,\omega]{R} u$ correspond $u \xrightarrow[i,\omega]{R^{-1}} t$ à laquelle est associée la transformation

$u \longrightarrow)_{i,\omega}\ u_1 \xrightarrow[i^*]{S^{-1}} t_1 \longrightarrow (_{i,\omega}\ t$ et cela de façon déterministe

Donc à toute règle de R^{-1} correspond une dérivation de S^{-1}.

(on généralise sans problème à un nombre quelconque de règles de R)

Réciproquement, soient t, u $\in T(\Sigma)$ et $t \xrightarrow[*]{S^{-1}} u$ une transformation de S^{-1}

Donc $u \xrightarrow[*]{S} t$ est définie et, puisque S simule R, correspond à une trans

formation $u \xrightarrow[*]{R} t$

Donc, dans R^{-1}, on a la transformation $t \xrightarrow[*]{R^{-1}} u$

Toute dérivation de S^{-1} basée sur $T(\Sigma)$ correspond donc à une transformation de R^{-1}.

Donc S^{-1} simule R^{-1}.

Démonstration du lemme 2 :

Montrons que toute dérivation de R u R^{-1} est simulée par une transformation de S u S^{-1} :

Soient t, u $\in T(\Sigma)$ tels que $t \xrightarrow[RuR^{-1}]{*} u$

On peut supposer sans nuire à la généralité que cette transformation est du type :

$$t \xrightarrow[i,\omega]{R} t_1 \xrightarrow[j,\omega']{R^{-1}} u$$

or $t \xrightarrow[i,\omega]{R} t_1$ est simulée dans S

$t_1 \xrightarrow[j,\omega']{R^{-1}} u$ est simulée dans S^{-1}

donc $t \xrightarrow[*]{RuR^{-1}} u$ est simulée dans S u S^{-1}

Montrons que, réciproquement, toute transformation de S u S^{-1} basée sur $T(\Sigma)$ est la simulation d'une transformation de R u R^{-1}.

Soient t, u $\in T(\Sigma)$ et $t \xrightarrow[SuS^{-1}]{*} u$

On peut supposer, sans nuire à la généralité, que l'on commence par appliquer une règle de S (forcément une ouverture de parenthèse). On a donc le schéma :

$$t \xrightarrow[(_{i,\omega}]{} t_1 \xrightarrow[SuS^{-1}]{*} u$$

Les règles de S^{-1} applicables après l'ouverture de $(_{i,\omega}$ sont, à cause du déterminisme, uniquement les règles inverses de la dernière règle de S appliquée.

Donc, soit les règles de S^{-1} n'effacent pas la parenthèse ouvrante et l'on peut considérer qu'on a seulement "reculé" dans la simulation de i, soit elles l'on effacé et ont commencé la simulation de l'inverse d'une règle de R. Dans ce cas, tout se passe comme si l'on avait débuté par cette simulation.

Donc cette transformation $t \xrightarrow[SuS^{-1}]{*} u$ peut-être décrite sous la forme d'une suite de simulations de règles de R ou R^{-1} successives, ce qui signifie exactement que toute dérivation de S u S^{-1} basée sur $T(\Sigma)$ est la simulation d'une dérivation de R u R^{-1}.

Démonstration du lemme 3 :

Montrons que toute transformation de R u R' peut être simulée dans S u S'. Soient t, u $\in T(\Sigma)$ tels que $t \xrightarrow[RuR']{*} u$, ce que l'on peut réécrire de la façon suivante :

$$t \xrightarrow[R]{i_1} t_1 \xrightarrow[R']{j_1} t_2 \xrightarrow[R]{i_2} \cdots \xrightarrow[R']{j_k} u$$

où certaines des règles i_ℓ, j_ℓ peuvent correspondre à l'identité.

Puisque chacune de ces étapes peut être simulée par S ou S', alors $t \xrightarrow[RuR']{*} u$ peut être simulé dans S u S'.

Réciproquement, soient t, u ε T(Σ) tels que $t \xrightarrow[SuS']{*} u$

On peut décomposer cette transformation comme précédemment en remarquant de plus que :

1) La première règle applicable engendre une ouverture de parenthèse

2) Les seules règles applicables alors correspondent à la simulation de la règle de R ou de R' indiçant la parenthèse ouvrante.

C'est-à-dire qu'après l'ouverture d'une parenthèse, on reste dans le système correspondant jusqu'à la fermeture de cette parenthèse. (Ceci est vrai car S et S' ont des alphabets auxiliaires disjoints par hypothèse).

Donc $t \xrightarrow[SuS']{*} u$ peut se réécrire sous la forme suivante :

$$t \longrightarrow t_1 \xrightarrow[S]{*} t_2 \longrightarrow t_3 \longrightarrow t_4 \xrightarrow[S']{*} t_5 \longrightarrow t_6 \ldots\longrightarrow u$$
$$\quad (_{i,\omega} \qquad)_{i,\omega} \quad (_{j,v} \qquad)_{j,v} \qquad)_{k,z}$$

où certaines des simulations peuvent correspondre à l'identité.

A chacune de ces phases successives, on peut associer suivant les cas une règle de R ou de R'.

Donc toute transformation de S u S' basée sur T(Σ) est la simulation d'une dérivation de R u R', ce qui termine la démonstration du lemme 3.

Ces trois lemmes nous permettent d'énoncer les résultats suivants :

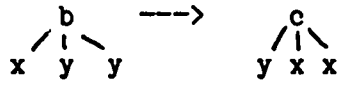
Corollaire 1 :

Soit R un système de réécriture complet sur T(Σ). Il existe deux systèmes de réécriture linéaires S_1 et S_2 tels que

1) S_1 et S_2 ont des alphabets de symboles auxiliaires disjoints

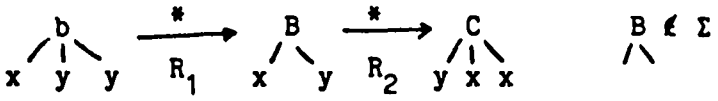
2) $S_1 \cup S_2^{-1}$ simule R

Idée de preuve :

A toute règle de R, par exemple 

, on associe

un nouveau terme et deux systèmes de réécriture R_1 et R_2 tels que :



R_1 et R_2 sont complets et linéaires à droite

donc, d'après les lemmes précédents :

si S_1 simule R_1 et S_2 simule R_2 $\Rightarrow$ $S_1 \cup S^{-1}$ simule $R_1 \cup R_2^{-1}$

Or on voit clairement que $R_1 \cup R_2^{-1}$ simule R, d'où le résultat.

(Remarque : on peut toujours définir S_1 et S_2 sur des alphabets auxiliaires disjoints).

Corollaire 2 :

Soit E une théorie équationnelle définie sur $T(\Sigma)$ complète, c'est-à-dire : $\forall\, t_i = t_i' \in E \quad V(t_i) = V(t_i')$

alors il existe un système de réécriture linéaire S tel que :

$S \cup S^{-1}$ simule E (i.e : $\forall (t,u) \in T(\Sigma)^2 \quad t \equiv u \Leftrightarrow t \xrightarrow[S \cup S^{-1}]{*} u$)

idée de preuve :

A toute équation de E, du type $b(x,y,y) = c(y,x,x)$ on associe un nouveau terme et deux systèmes de réécriture linéaires droits complets.

$b(x,y,y) \xrightarrow{R_1} B(x,y) \xleftarrow{R_2} C(y,x,x)$ $\quad B \notin \Sigma$

Posons $R = R_1 \cup R_2$. R est linéaire droit.

Soit S linéaire simulant R.

D'après le lemme 1, $S \cup S^{-1}$ simule $R \cup R^{-1}$, qui lui-même simule E

d'où le résultat.

CHAPITRE VII

Complexité

Complexité

Introduction :

Dans ce chapitre, nous introduisons la notion de complexité d'un système de réécriture. Cela permettra de comparer des systèmes de réécriture, et en particulier d'évaluer la différence existant entre les systèmes de réécriture linéaires et les autres.

Nous avons constaté dans les chapitres précédents que la simulation d'un système quelconque par un système de réécriture linéaire ne se fait qu'au prix d'un accroissement considérable du nombre de règles et de symboles fonctionnels et du contrôle de l'ordre des applications de règles (perte du non déterminisme).

Nous essaierons ici de quantifier cette différence.

Dans un premier temps, nous définissons la profondeur d'une transformation, puis nous donnons une autre caractérisation peut-être plus parlante.

Ayant montré dans un deuxième temps que l'on peut accomplir, grâce à des systèmes de réécriture de complexité finie, certaines opérations élémentaires sur les arbres, nous les composerons finiment, obtenant ainsi avec un système de complexité finie et partant d'un arbre quelconque, la famille des termes récursivement énumérables.

Comme d'autre part, nous montrons que nos simulations sont de complexité infinie, cela nous permettra la distance qui sépare les systèmes linéaires des autres.

Profondeur d'une dérivation, d'une transformation :

Définitions :

1. Dérivation :

Soit R système de réécriture sur $T(\Sigma)$ et $(t,u) \in T(\Sigma)$.

$t \underset{R}{\overset{*}{\vdash}} u$ est une dérivation s'il existe n règles $i_0, \ldots i_{n-1}$ de R telles que $t_0 = t \underset{}{\overset{i_0}{\vdash}} t_1 \ldots \underset{}{\overset{i_{n-1}}{\vdash}} t_n = u$

2. Transformation :

Soit R système de réécriture sur $T(\Sigma)$, et $(t,u) \in T(\Sigma)$.

Alors (t,u) est une transformation dans R s'il existe au moins une dérivation de R telle que $t \underset{R}{\overset{*}{\vdash}} u$.

(On peut considérer que la transformation (t,u) est la classe de toutes les dérivations de R telles que $t \underset{R}{\overset{*}{\vdash}} u$.)

3. Profondeur d'une dérivation :

Soit R système de réécriture sur $T(\Sigma)$, et $t \underset{R}{\overset{*}{\vdash}} u$ une dérivation.

On définit récursivement la profondeur de cette dérivation, notée $\Pi_R(t \overset{*}{\vdash} u)$, de la façon suivante :

- Si $t \underset{R}{\overset{0}{\vdash}} u = t$ (aucune règle n'est appliquée) alors $\Pi_R(t \overset{0}{\vdash} u) = 0$
- Soit une dérivation quelconque $t \overset{*}{\vdash} u$

  Alors $\Pi_R(t \overset{*}{\vdash} u) \leq n + 1$ ssi :

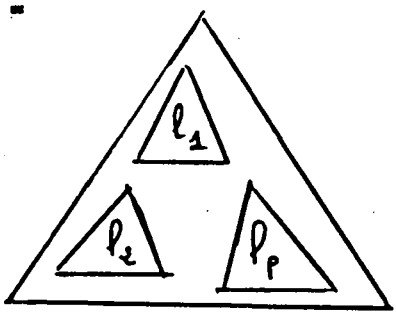
  Il existe deux dérivations $t \overset{*}{\vdash} t'$

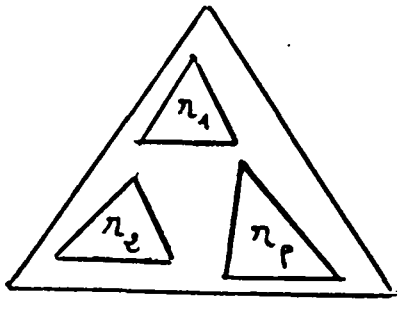
  avec $\Pi_R(t \overset{*}{\vdash} t') \leq n$

  et $t' \overset{*}{\vdash} u$ de telle façon que :

  $t' =$ $\qquad$ $u =$

où :





Chaque $\ell_i \longrightarrow r_i$, $i \in [1,p] \in R$ et son application dans t' ne chevauch
celle d'aucune règle $\ell_j \longrightarrow r_j$ $j \in [1,p]$

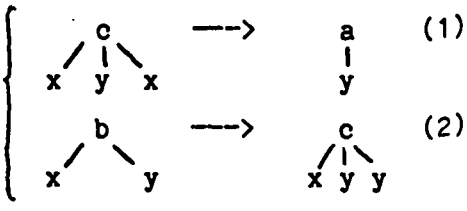
Aucune de ces p règles n'était applicable avant d'avoir atteint l'arbr
t'.

Informellement, une dérivation est de profondeur 1 si les règles à appli
quer peuvent l'être dans un ordre quelconque (donc en parallèle).

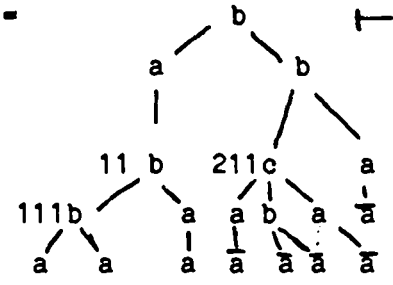
Exemples :

a. Dérivation de profondeur 1 :

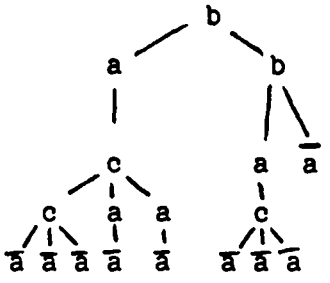
Soit $\Sigma = \{a \quad b \quad c \quad \bar{a}\}$ et $R =$



t =



$\vdash\!\!\!\xrightarrow{*}$ u =

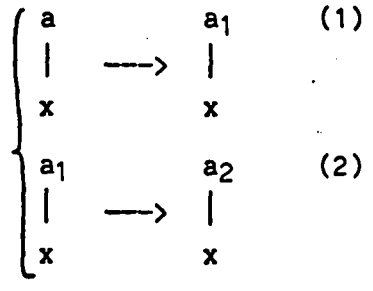


en appliquant (1) aux occurences 11, 111, 2112 de t, et (2) en 211.
Cette dérivation est de profondeur 1 car l'ordre dans lequel ces quatre
applications de règles sont effectuées n'influe pas sur le résultat.

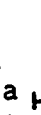
b. Dérivation de profondeur 2 :

Soit $\Sigma = \{a \quad a_1 \quad a_2 \quad \bar{a}\}$

et $R =$



$\Pi_R\left(a \vdash\!\!\!\xrightarrow{*} a_2\right) = 2$ car l'ordre dans lequel les deux règles doivent être



appliquées est ici fixé.

Définition : Profondeur d'une transformation

Soit R système de réécriture sur $T(\Sigma)$ et (t,u) une transformation de R.

On appelle profondeur de la transformation (t,u), notée $\Pi_R(t,u)$, la quantité définie par :

$$\Pi_R(t,u) = \inf_{t \overset{*}{\underset{R}{\longleftarrow}} u} \{\Pi_R(t \overset{*}{\longleftarrow} u)\}$$

Remarque : Puisque $\Pi_R \geq 0$, cet inf est atteint pour une certaine dérivation "minimale" notée $t \overset{*}{\underset{m}{\longleftarrow}} u$

La profondeur d'une transformation correspond intuitivement au nombre minimal de "passages" à effectuer pour dériver t en u. (Un "passage" étant en fait une dérivation de profondeur 1).
Nous allons donner une autre caractérisation de la profondeur d'une transformation, grâce au lemme suivant :

Définition préliminaire :

Soit $t \overset{*}{\longleftarrow} u$ une dérivation dans R.
On dira que deux applications de règles sont liées dans $t \overset{*}{\longleftarrow} u$ si elles interviennent dans la dérivation successivement et dans un ordre fixé. On appellera chaîne d'applications de règles liées (ou plus simplement chaîne de règles liées) toute suite d'applications de règles $i_0, i_1, \ldots i_n$ telle que chaque couple (i_k, i_{k+1}) soit composé de deux règles liées (i_k devant être appliquée avant i_{k+1}) dans $t \overset{*}{\longleftarrow} u$.

Lemme :
Une transformation (t,u) dans R est de profondeur n ssi :
dans la dérivation $t \overset{*}{\underset{m}{\longleftarrow}} u$, il existe une chaîne de n règles liées, et

aucune chaîne de longueur supérieure à n.

Démonstration :

$\Pi_R(t,u) = n \iff \Pi_R(t \overset{*}{\underset{m}{\longrightarrow}} u) = n$ par définition.
Par définition de la profondeur d'une dérivation, on en conclut qu'il existe dans cette dérivation $t \overset{*}{\underset{m}{\longrightarrow}} u$ n applications de règles à appliquer successivement dans un ordre précis. Ce qui signifie exactement qu'il existe dans $t \overset{*}{\underset{m}{\longrightarrow}} u$ une chaîne de n règles liées.

Montrons qu'il n'y a pas de chaîne plus longue :
Par définition de la profondeur d'une dérivation, s'il existait dans $t \underset{m}{\overset{*}{\longleftarrow}} u$ une chaîne de règles liées de longueur > n, alors $\Pi_R(t \underset{m}{\overset{*}{\longleftarrow}} u) > n$

donc $\Pi_R(t,u) = \Pi_R(t \underset{m}{\overset{*}{\longleftarrow}} u) > n$

ce qui est contraire à l'hypothèse.

Complexité d'un système de réécriture :

Définition : Soit R système de réécriture.
On appelle complexité de R, notée $\mathcal{C}(R)$, la quantité définie de la manière suivante :

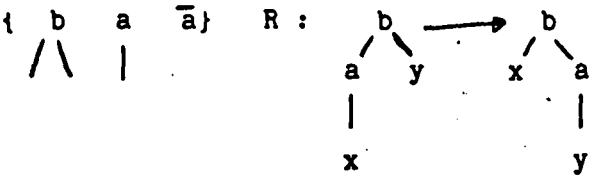
$$\mathcal{C}(R) = \sup_{(t,u) \in R} \{\Pi_R(t,u)\}$$

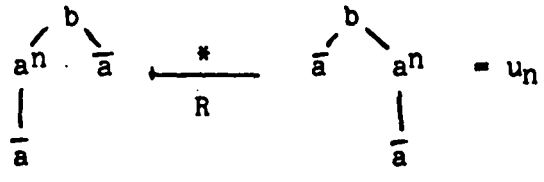
D'après le lemme précédent, $\mathcal{C}(R)$ est aussi égale à la borne supérieure des chaînes de règles liées pour R.

Exemples et résultats :

Soit R système de réécriture sur $T(\Sigma)$
S'il existe $t \in T(\Sigma)$ tel que t se dérive par R en une infinité de termes, alors $\mathcal{C}(R) = +\infty$.

En effet : si $\mathcal{C}(R) < +\infty$, la longueur des chaînes liées dans R est bornée. Puisque R et t sont finis, il en résulte que t n'a qu'un nombre fini de descendants.

La réciproque est fausse.
Considérons pour le montrer $\Sigma = \{b, a, \bar{a}\}$ R : 

Tout arbre a un nombre fini de descendants, égal au nombre de "a" descendants gauches d'un b.
Pourtant $\mathcal{C}(R) = +\infty$
Considérons en effet $t_n =$  $= u_n$

Cette dérivation (qui est minimale car unique) est clairement de profondeur n donc $\mathcal{C}(R) \geq \sup_{n \in N} \{\Pi_R(t_n, u_n)\} = +\infty$

Définition :

Soit R système de réécriture sur $T(\Sigma)$. On définit sur $T(\Sigma)$ la relation

$$\hat{R} = \{(t,t') / t \overset{*}{\underset{R}{\longrightarrow}} t'\}$$

R est donc l'ensemble des transformations.

Définition :

Soient R, R' deux systèmes de réécriture, on dira que :

$$R \backsimeq R' \text{ ssi } \hat{R} = \hat{R}'$$

Définition :

Soit A une relation sur $T(\Sigma)$ réalisable par un système de réécriture (ce qui équivaut à dire que A est une relation récursivement énumérable).

On définit : $\mathcal{C}(A) = \inf\{\mathcal{C}(R) / R = A\}$

$\mathcal{C}(n) = \{A \text{ relation sur } T(\Sigma) / \ \mathcal{C}(A) = n\}$

On obtient de cette façon une hiérarchie infinie de relations, comme l'ont montré Engelfriet et B. Baker.

Théorème :

$\forall n \in N \quad \mathcal{C}(n) \subsetneq \mathcal{C}(+\infty)$.

Démonstration :

Il suffit de prendre une relation A telle que :

$\exists t \in T(\Sigma)$ tel que $\{(t,u) \in A\}$ soit infini.

En effet, quelque soit R système de réécriture vérifiant $\hat{R} = A$, $\mathcal{C}(R) = +\infty$ donc aucun système de complexité finie ne réalisera A.

De même, on a le résultat suivant :

Théorème :

Il existe des relations récursivement énumérables A telles que :

pour tout terme t de $T(\Sigma)$ $\{(t,u) \in A\}$ soit fini et $\mathcal{C}(A) = +\infty$

Démonstration :

Il suffit de considérer l'exemple de la page précédente.

Nous allons montrer maintenant que certaines transformations usuelles d'arbres peuvent être réalisées par des systèmes de réécriture de "petite complexité".

Proposition 1 :

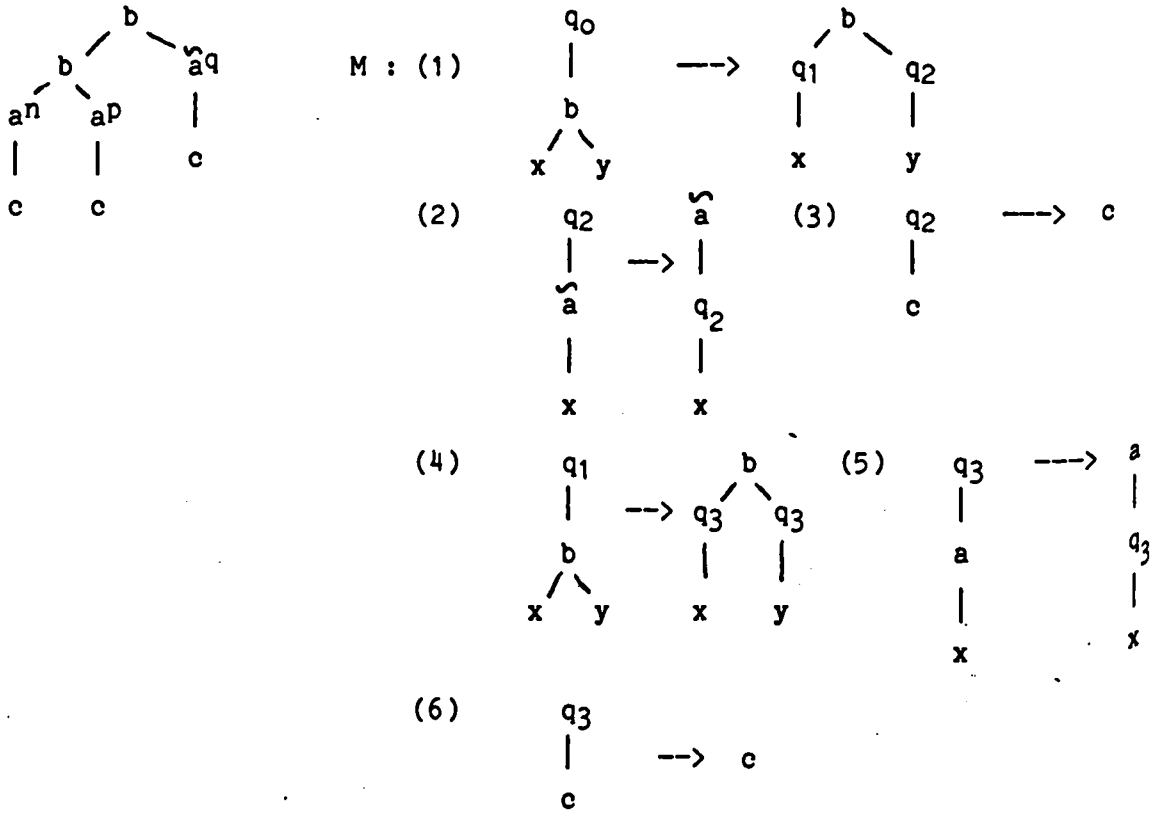
Soit M = (Σ, Q, F, R) un automate fini d'arbre.

Il existe un algorithme permettant de définir R_m, système de réécriture de complexité 2 défini sur $\Sigma \cup Q \cup \overline{\Sigma} \cup \{\#\}$ tel que

$$\forall\, t \in T(\Sigma)^1_0 \quad t \in \mathcal{L}(M) \iff (t, \bar{t}^{\#}) \in \hat{R}_m$$

Nous illustrons ce résultat sur un exemple :

Soit M l'automate d'arbre reconnaissant les termes de la forme :



Ces six règles définissent sur $T(\Sigma \cup Q)$ un systèmes de réécriture de complexité infinie (il faut suivre pas à pas la descente des q_i).

On peut cependant le simuler par un système de réécriture de complexité finie défini sur $\Sigma \cup \overline{\Sigma} \cup \{\#\} \cup Q$:

(1) $\#(b(x,y)) \longrightarrow \overline{b}(q_1(x), q_2(y))$ (2) $b(x,y) \longrightarrow q_1(\overline{b}(q_3(x), q_3(y)))$

(3) $\hat{a}(x) \longrightarrow q_2(\underline{\hat{a}}(q_2(x)))$ (4) $a(x) \longrightarrow q_3(\overline{a}(q_3(x)))$

(5) $c \longrightarrow q_2(\overline{c})$ (6) $c \longrightarrow q_3(\overline{c})$ (7) $\forall i \in \{1,2,3\}$ $q_i(x) \longrightarrow x$

Ce système de réécriture appliqué à un arbre de la forme $\#(t)$ est noethérien mais non confluent (à cause des règles (5) et (6))

Il "barre" tous les symboles de t et intercale entre chaque couple de symboles successifs une suite $q_i(q_j)$ (si ces symboles sont dans $\{b, a, c, \hat{a}\}$),

puis efface les suites $q_i(q_i)$

Donc si $t \in \mathcal{T}(M)$, il existe une dérivation $\#(t) \xrightarrow[R_M]{*} u$ telle que $u = \overline{t}$.

Réciproquement, si $t \notin \mathcal{T}(M)$, toute dérivation $\#(t) \xrightarrow[R_M]{*} u$ sera telle qu'il existera dans u une suite $q_i(q_j)$, $i \neq j$.

On peut remarquer que R_M est linéaire.

D'autre part, $\mathcal{C}(R_M) = 2$ pour $t \in T(\Sigma)$.

En effet, les seules chaînes de règles liées que l'on peut obtenir font intervenir une règle du type (1) à (6) suivie de la règle (7), ceci grâce au "barrage" des lettres.

Proposition 2 :

On peut réaliser un homomorphisme Ψ (resp. linéaire) par un système de réécriture R_Ψ (resp. linéaire) tel que $\mathcal{C}(R_\Psi) = 1$.

Démonstration :

Ici encore, nous ne ferons qu'illustrer ce résultat sur un exemple :

Soit Ψ $\left\{\begin{array}{l} a \longrightarrow \bar{a}(x_1) \\ x_1 \\ b \longrightarrow x_1\ c\ x_2 \\ \bar{a} \longrightarrow \bar{a} \end{array}\right.$ alors R_Ψ = $\left\{\begin{array}{l} a \longrightarrow \hat{\bar{a}}(x_1) \\ x \\ b \longrightarrow x_1\ \hat{c}\ x_2 \\ \bar{a} \longrightarrow \hat{\bar{a}} \end{array}\right.$

En notant $\hat{u}$ l'arbre $u \in T(\Sigma)$ dont tous les symboles sont surmontés d'un '$\wedge$', on a clairement :

$$(t, \hat{u}) \text{ est une tranformation pour } R_\Psi \Leftrightarrow \Psi(t) = u$$

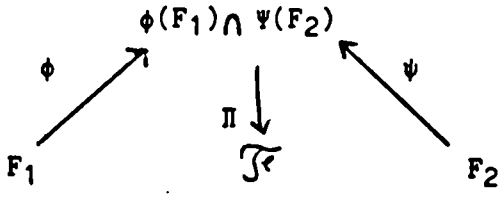
R_Ψ est de complexité 1, car le marquage des symboles fonctionnels en partie droite empêche leur chevauchement.

Nous utiliserons ces résultats en liaison avec ceux obtenus par J. Mongy et plus particulièrement :

Il existe un algorithme qui, à toute forêt $\mathcal{F}$ récursivement énumérable (donnée par une grammaire d'arbres quelconque), associe trois homomorphismes Π, ϕ Ψ et deux forêts reconnaissables F_1 et F_2 tels que :

$$\mathcal{F} = \Pi\,(\phi(F_1) \cap \Psi(F_2))$$

ce que l'on peut figurer sur le schéma suivant :



Nous pouvons, grâce aux propositions 1 et 2, exprimer ce résultat en termes de systèmes de réécriture :

Soient Σ_1, Σ_2, Δ, Γ quatre alphabets gradués où $\Sigma_1 \cap \Sigma_2 = \emptyset$

F_1 forêt reconnaissable définie sur $T(\Sigma_1)$

F_2 forêt reconnaissable définie sur $T(\Sigma_2)$

$\phi : \Sigma_1 \longrightarrow \Delta$

$\psi : \Sigma_2 \longrightarrow \Delta$

$\Pi : \Delta \longrightarrow \Gamma$

F :définie sur Γ

D'après la proposition 1, il existe R_1 associé à F_1, R_2 associé à F_2

$\mathcal{C}(R_1) = \mathcal{C}(R_2) = 2$

ϕ, ψ, Π peuvent être simulés respectivement par R_ϕ, R_ψ, R_Π de complexité 1, de par la proposition 2.

Il nous reste à représenter sous forme de système de réécriture l'intersection $\phi(F_1) \cap \psi(F_2)$.

Pour cela, on utilise un nouveau symbole fonctionnel

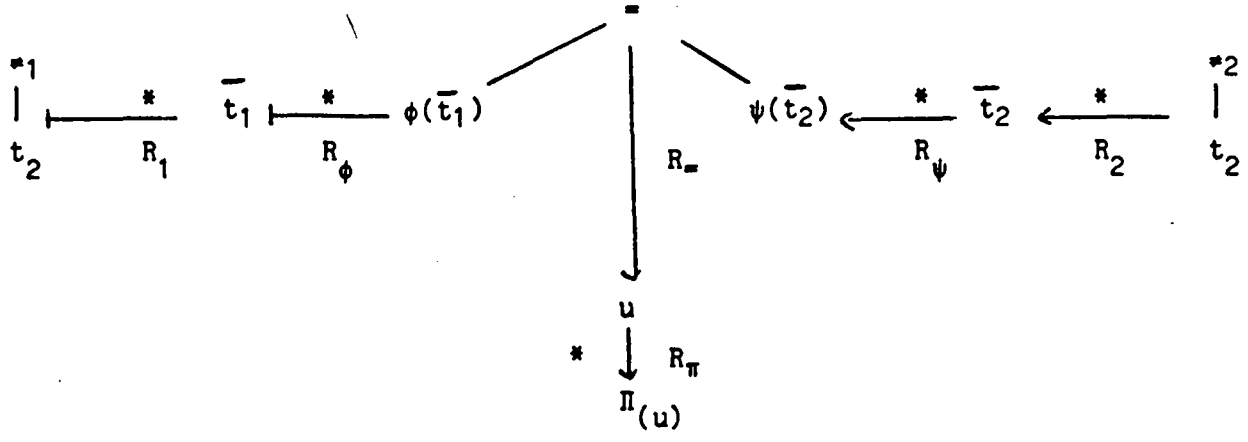
$= \in \Delta$, $= \notin \Sigma_1 \cup \Sigma_2 \cup \Gamma$

tel que : $\Pi(=)$ est l'arbre vide.

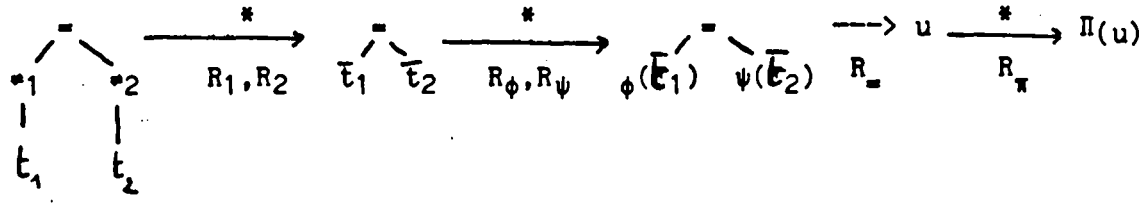
$$= \begin{array}{c} / \backslash \\ x\ y \end{array} \longrightarrow x \quad \text{(système de réécriture } R_= \text{ de complexité 1)}$$

$$= \begin{array}{c} / \backslash \\ x\ x \end{array}$$

On obtient alors le schéma suivant, pour $(t_1, t_2) \in F_1 \times F_2$:



que l'on peut aussi exprimer :



Le test d'égalité composant $R_=$ ne s'effectuera que lorsque t_1 et t_2 auront été entièrement transposés dans l'alphabet Δ.
Les systèmes de réécriture lors de cette transformation se succèdent de façon précise : l'un n'intervenant que lorsque les termes concernés ont été entièrement transposés (par le système précédent) dans l'alphabet idoine.
On peut donc pour calculer la complexité du système réunion de ces systèmes, additionner les complexités de chacun d'eux.
On déduit alors de cette construction le théorème suivant :

Théorème :

Par un système de réécriture de complexité 5, on peut engendrer, à partir du monoïde libre $T(\Sigma)$, toute forêt récursivement énumérable.

dem : voir construction précédente.

Ce résultat prend toute son importance lorsqu'on le compare aux suivants, concernant les systèmes de réécriture linéaires.

Théorème (de simulation inverse) :

Tout système de réécriture R tel que $\mathcal{C}(R) = 1$ peut être réalisé par un bimorphisme B.

(de plus, si R est linéaire, B l'est aussi).

Idée de la preuve :

Soit $R = \{(\ell_i \longrightarrow d_i),\ i \in [1,n]\}$ défini sur $T(\Sigma)$

On associe à chaque règle i un nouveau symbole fonctionnel ρ_i et un terme :

où les x_j $j \in [1,n_i]$ sont les variables apparaissant dans ℓ_i.

Soit $\Sigma' = \Sigma \cup \{\rho_i,\ i \in [1,n]\}$

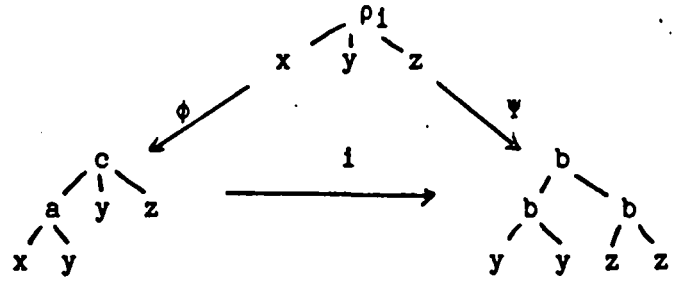
On définit les deux morphismes $\Psi, \phi : \Sigma' \longrightarrow \Sigma$

de telle façon que $\forall\ x \in \Sigma \quad \phi(x) = \Psi(x) = x$

$$\forall\ i \in [1,n] \quad \phi(\rho_i) = \ell_i$$

$$\Psi(\rho_i) = d_i$$

Exemple :



Remarque :

La condition $\mathcal{C}(R) = 1$ est nécessaire : elle permet de considérer chaque règle $\ell_i \longrightarrow d_i$ indépendamment (il n'y a pas de chaînes de règles liées).

Corollaire :

Soit R système de réécriture linéaire à droite tel que $\mathcal{C}(R) < +\infty$ alors R conserve la reconnaissabilité.

Idée de la preuve :

Nous utilisons le résultat suivant :

Soit Rec la classe des forêts reconnaissables

Rec est close par intersection, homomorphisme linéaire et homomorphisme inverse.

Soit $n = \mathcal{C}(R)$.

donc $\forall (t,u) \in \hat{R}$, $\exists\ t_0 = t, t_1, \ldots, t_n = u$ tels que :

$$t = t_0 \xrightarrow{*} t_1 \xrightarrow{*} t_2 \ldots t_{n-1} \xrightarrow{*} t_n = u$$

Chacune de ces dérivations est de complexité inférieure ou égale à 1.

Ainsi chaque pas $t_i \xrightarrow{*} t_{i+1}$ peut être réalisé par un bimorphisme B_i.

Donc $B_n \circ B_{n-1} \circ \ldots \circ B_1 = B$

Puisque les bimorphismes conservent la reconnaissabilité, il en est donc de même pour R.

Conclusion :

La notion de complexité d'un système de réécriture et les résultats la concernant permettent d'apprécier la différence existant entre les systèmes de réécriture linéaires et les autres.

Si un système non linéaire de complexité finie permet d'engendrer, à partir de 2 forêts reconnaissables, toute forêt récursivement énumérable, il n'en est pas de même pour un système linéaire de complexité finie.

Dans le cas général, la simulation d'un système non linéaire par un système linéaire ne se fait qu'au prix d'une "complication infinie". Ceci même dans le cas relativement simple (il s'agit uniquement de copier) de la linéarisation à droite.

Les systèmes de réécriture non linéaires permettent de définir de façon peu complexe des tâches infiniment complexes pour un système linéaire.

BIBLIOGRAPHIE

Bibliographie

B.S. BAKER
Tree transductions and families of tree-languages 5 th ACM Proc. on Theory of Computing (1973)

M. DAUCHET
Transductions de forêts Bimorphismes de magmoïdes.
Doctorat d'Etat LILLE I (1977)

N. DERSHOWITZ
Termination of linear rewriting systems-Preliminary version. In Automata, Languages and Programming, 8th Coll. Israël (Eds. S. Even et O. Kariv), LNCS 115, Springer Verlay, New York 1981

N. DERSHOWITZ
Orderings for term rewriting systems.
Proc. 20th Symposium on Foundations of Computer Science pp 123-131 (1979)

N. DERSHOWITZ et Z. MANNA
Proving termination with multiset ordering.
Commun. ACM 22,8 (Août 1979) 465-476

J. ENGELFRIET
Top-down tree transducers with regular look-ahead. Math. Systems Theory 10, pp 289-303 (1977)

J. ENGELDRIET
Bottom-up and Top-down tree transformation. A comparison. Math. Systems Theory 9, pp 198-231 (1975)

J.V. GUTTAG, D. KAPUR et D.R. MUSSER
Derived Pairs, Overlap Closure and Rewrite Rules. New tools for Analysing term rewriting systems in Automata, Languages, and Programming, 9 coll. Aarhus, Danemark (Juillet 1982) LNCS 140, Springer Verlag.

J.V. GUTTAG, D. KAPUR et D.R. MUSSER
On proving uniform termination and restricted termination of rewriting systems. Tech. Rep. n°81CRD 272, General Electric Company (Nov. 1981)

G. HUET
Confluent Reductions : Abstract properties and applications to term rewriting systems. J. ACM 27 (1980) pp 797-821

G. HUET et D.S. LANKFORD
On the uniform halting problem for term rewriting systems. Lab. Rep, n° 283, INRIA Le Chesnay, FRANCE (Mars 1978)

G. HUET et D.C. OPPEN
Equations and rewrite rules : A survey. Formal Language Theory : Perspectives and Open Problems (Ed : R. Book) Academic Press (1980)

D. KNUTH et P. BENDIX
Simple word problems in universal algebras. In Computational Problems in Abstract Algebra
(Ed : J. Leech) Pergamon Press, Elmsford, New-York. pp 263-297 (1970)

J. MONGY- STEEN
Transformations de noyaux reconnaissables d'arbres. Forêts Rateg. Thèse de 3ème cycle LILLE I (1981)

G.E. PETERSON et M.E. STICKEL
Complete set of reductions for equational theories with complete unification algorithms. Tech. Rep., Dep. of Computer Science, University of Arizona, Tucson, Ariz. (Sept. 1977)

A. PETTOROSSI
Comparing and patting together Recursive Path Ordering, Simplification Orderings and non-ascending property for termination proofs of term rewriting systems. In Automata, Language, and Programming, 8th Coll., Israël (Eds : S. Even et O. Kariv) LNCS 115, Springer Verlay, N.Y. (1981)

D. PLAISTED
Well-founded Orderings for proving termination of systems of rewrite rules. Tech. Rep. 78-932, Dep. of Computer Science, Univ. of Illinois, Urbana-Champaign, Ill (Juillet 1978)

D. PLAISTED
A recursively defined ordering for prowing termination of term rewriting systems. Tech. Rep 78-943, Dep. of Computer Science, Univ. of Illinois, Urbana-Champaign, Ill. (Sept. 1978)

F. REINIG
L'ordre de décomposition : un outil incremental pour prouver la terminaison finie de systèmes de réécriture de termes.
Thèse de 3ème cycle Nancy I (1981)

B.K. ROSEN
Tree-manipulating systems and Church-Rosser Theorems. J. ACM 20,1 (Janvier 1973) pp 160-187.